Universidad Nacional Autónoma de México

Facultad de Ciencias

El ISOMORFISMO DE CURRY HOWARD, UN FUNDAMENTO LÓGICO PARA LA PROGRAMACIÓN FUNCIONAL

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
EDUARDO GERÓNIMO PACHECO GÓMEZ

DIRECTOR DE TESIS:
DR. FAVIO EZEQUIEL MIRANDA PEREA

2008

Hoja de Datos del Jurado

1. Datos del alumno
   Pacheco
   Gómez
   Eduardo Gerónimo
   58 50 24 57
   Universidad Nacional Autónoma de México
   Facultad de Ciencias
   Ciencias de la computación
   301048938
2. Datos del tutor
   Dr
   Favio Ezequiel
   Miranda
   Perea
3. Datos del sinodal 1
   Dr
   Carlos
   Torres
   Alcaraz
4. Datos del sinodal 2
   Dr
   Francisco
   Hernández
   Quiroz
5. Datos del sinodal 3
   M en C
   Araceli Liliana
   Reyes
   Cabello
6. Datos del sinodal 4
   Lic en C C
   Lourdes del Carmen
   González
   Huesca

Índice general

Introducción **v**

1. Lógica intuicionista **1**
1.1. Lógica proposicional intuicionista 2
1.2. Semántica 2
1.3. La interpretación BHK 3
1.4. Deducción natural 5
1.4.1. Significado de conectivos 6
1.4.2. El sistema DN 7
1.4.3. Reglas para la negación 9
1.5. Normalización de demostraciones 11

2. Sistemas de tipos **15**
2.1. Cálculo lambda puro 16
2.1.1. Sintaxis del cálculo lambda 17
2.1.2. α-equivalencia 19
2.1.3. Substitución 20
2.1.4. β-reducción 21
2.1.5. No terminación del cálculo lambda 21
2.2. Expresividad del cálculo lambda 22
2.2.1. Booleanos 22
2.2.2. Pares 23
2.2.3. Números naturales 23
2.2.4. Teorema de Church-Rosser 24
2.3. Cálculo lambda con tipos simples 24
2.3.1. Cálculo lambda con tipos *a la Curry* 25
2.3.2. Cálculo lambda con tipos *a la Church* 28
2.3.3. Problemas de decisión 30
2.4. Normalización de la β-reducción 31
2.5. Extensión con sumas y productos 31
2.5.1. Reducción 33

3. El isomorfismo de Curry-Howard **35**
3.1. El isomorfismo de Curry-Howard 36
3.2. Contenido computacional y codificación de las derivaciones . . . 39
3.2.1. La implicación 40
3.2.2. La conjunción 40
3.2.3. La disyunción 41
3.2.4. El falso 41
3.3. Normalización y $\beta-$reducción 43
3.4. Curry-Howard y programación 45
3.4.1. Numerales de Church 46
3.5. Resumen 46

4. MinHaskell **49**
4.1. Programación funcional 49
4.2. Sintaxis 50
4.2.1. Sintaxis concreta 51
4.2.2. Sintaxis abstracta 52
4.3. Semántica 53
4.3.1. Semántica dinámica 54
4.3.2. Semántica estática 56
4.4. Seguridad del lenguaje 58
4.4.1. Terminación 58
4.5. Rompiendo el isomorfismo 59
4.6. Implementación 60
4.6.1. Análisis léxico 60
4.6.2. Análisis sintáctico 61
4.6.3. Substitución 62
4.6.4. Verificación de tipos 65
4.6.5. Evaluación 65
4.7. Ejemplos y ejecución 66

5. Conclusiones **67**

Introducción

En 1960 E. P. Wigner, ganador del premio Nobel de física en 1963, publicó un artículo acerca de la misteriosa efectividad que las matemáticas tenían para fundamentar y modelar fenómenos de las ciencias naturales (véase [6]). A través de numerosos ejemplos mostró que una formulación matemática de los fenómenos físicos es apropiada y exacta atreviéndose a afirmar que las matemáticas son el lenguaje correcto para formular las leyes de la naturaleza y puntualizando que las razones de este éxito no estaban completamente entendidas aventurándose a declarar que esta efectividad es misteriosa y no tiene una explicación racional. Veinte años más tarde R. W. Hamming, ganador del premio Turing para Ciencias de la Computación otorgado por la Association for Computing Machinery (ACM), publicó un artículo relacionado (véase[27]). En este artículo proporciona más ejemplos que ponen de manifiesto la efectividad de las matemáticas en las ciencias naturales. Más aún, intentó responder a la pregunta de Wigner ¿Por qué son las matemáticas irrazonablemente efectivas? y, si bien, dio unas respuestas parciales, su conclusión fue que la pregunta permanece esencialmente sin contestar.

Desde entonces las ciencias de la computación han tenido un desarrollo espectacular, y tal como sucedió con las ciencias naturales, las matemáticas han jugado un papel de gran importancia. Diversas áreas de las matemáticas, incluyendo álgebra lineal, teoría de los números, probabilidad, teoría de gráficas y combinatoria han proporcionado instrumentos indispensables para el desarrollo de las ciencias computacionales. Pero es otra área de las matemáticas en particular la que nos interesa, la lógica matemática. De hecho la lógica ha resultado ser más efectiva e influyente en ciencias de la computación que en matemáticas en los últimos años, lo cual es curioso dado que gran parte del desarrollo de la lógica durante los últimos cien años surgió para confrontar la crisis de los fundamentos de las matemáticas a inicios del siglo XX.

Hoy, la lógica matemática es un área de investigación madura y altamente sofisticada que tiene numerosas aplicaciones en ciertas áreas de la matemática, pero que en los últimos cuarenta años ha tenido más relevancia en ciencias de la computación que en matemáticas. Los conceptos y métodos de la lógica ocupan un lugar central en la computación hasta el punto de que algunos la han llamado *el cálculo de las ciencias de la computación* (ver [20]).

Las relaciones y aplicaciones entre ambas disciplinas son múltiples, por ejemplo, es posible capturar clases de complejidad computacional mediante ciertas

lógicas; el uso de lógica de primer orden ha sido de gran utilidad como un lenguaje de consulta en bases de datos (véase [30]); el razonamiento acerca de los sistemas multiagente se sirve fructíferamente de lógica epistémica, etcétera.

En este trabajo nos interesa otra relación, a saber, la existente entre la teoría de tipos (cálculo lambda) y los fundamentos de los lenguajes de programación. En la década de los ochenta y noventa del siglo pasado el estudio de lenguajes de programación sufrió una revolución debido a la confluencia de ideas provenientes tanto de la lógica matemática y filosófica como de la ciencia de la computación teórica. La teoría de tipos emergió como un marco unificador conceptual para el diseño, análisis e implementación de lenguajes de programación. Los sistemas de tipos vierten luz sobre conceptos computacionales como abstracción de datos, polimorfismo y herencia. Así mismo proporcionan un fundamento para desarrollar lógicas de comportamiento de programas esenciales para razonar sobre los mismos. Además sugieren nuevas técnicas para implementar compiladores que mejoran la eficiencia e integridad del código generado.

La profunda relación entre lógicas y sistemas de tipos surgió de la observación de que el constructor de tipos exhibe una profunda e intrigante similitud con las reglas de introducción y eliminación para la implicación en el sistema de deducción natural de Gentzen. Esta observación se ha formalizado en lo que hoy se conoce como correspondencia o *isomorfismo de Curry-Howard*, el cual en primera instancia relaciona dos formalismos independientes: el cálculo lambda, introducido por Alonzo Church con el propósito de fundamentar a las matemáticas a partir del concepto de función, y los sistemas de deducción natural, introducidos por Gerard Gentzen como una formalización del proceso lógico deductivo cercano al razonamiento humano (ver [11]).
El propósito principal de este trabajo es describir detalladamente este isomorfismo, así como mostrar su aplicación mediante la implementación de un mini lenguaje funcional el cual representa al núcleo de todo lenguaje funcional real, por esto es que consideramos al isomorfismo de Curry-Howard como un fundamento para la programación funcional.

En el capítulo uno, se definirá el sistema básico de deducción natural de Gentzen para la Lógica Proposicional Intuicionista. En el capítulo dos, se dará una breve introducción a los sistemas de tipos, sus características y ventajas, lo que nos permitirá introducir al cálculo lambda puro, del cual se discutirán algunas propiedades importantes como su relación con los programas computacionales. Se discutirá el cálculo lambda con tipos simples, en sus dos presentaciones *a la Curry* y *a la Church*, los cuales inducen dos paradigmas distintos de programación, de los cuales hablaremos brevemente. Daremos además una extensión del sistema de tipos del cálculo lambda con tipos simples *a la Church*. El tema central de la tesis se discutirá en el capítulo tres. Se dará una prueba formal de la relación entre el sistema de deducción natural para la Lógica Proposicional Intuicionista y el cálculo lambda con tipos extendidos *a la Church*, tanto al nivel de deducciones como al de normalización de las mismas. Además se dará una interpretación computacional de las reglas del sistema de deducción natural la cual permitirá en el capítulo cuarto presentar de manera formal la descripción de un pequeño lenguaje de programación al cual hemos llamado minHaskell, además

presentaremos detalles de su implementación. Todo esto nos permitirá justificar porque el *isomorfismo de Curry-Howard* puede ser considerado un fundamento para la programación funcional.

Capítulo 1

Lógica intuicionista

No hay un sistema formal que pueda expresar la riqueza y complejidad del pensamiento humano, por lo que toda lógica sólo puede ser usada como una herramienta de propósito límitado, y no como un oráculo que pueda dar respuesta a todas las interrogantes. Por ejemplo, los principios de la lógica clásica han sido extremadamente útiles para describir y clasificar muchos de los patrones del razonamiento que ocurren en las matemáticas y en el diario acontecer, sin embargo, éstos no son los únicos posibles principios de razonamiento.

La lógica clásica está basada en la noción de verdad donde la veracidad de una proposición es una propiedad constante que es independiente de cualquier razonamiento, entendimiento o acción alguna. Así una proposición bien formada y sin ambigüedad debe ser verdadera o falsa, podamos o no demostrala, por lo que el significado de falso es igual a no verdadero, ésto queda expresado en el principio del tercer excluido, que establece que $A \vee \neg A$ siempre es verdadero para cualquier proposición A. Esto sin embargo genera ciertas desventajas desde un punto de vista pragmático. Veamos un ejemplo para ilustrar ésto. Considérese el siguiente enunciado:

Existen dos números irracionales, x, e y, tales que x^y es racional

Del cual podemos dar la siguiente demostración: si $\sqrt{2}^{\sqrt{2}}$ es racional entonces $x = y = \sqrt{2}$. Si no, entonces $x = \sqrt{2}^{\sqrt{2}}$, $y = \sqrt{2}$.
El problema con esta demostración es que no sabemos cual de las dos posibilidades es la correcta. Ahora consideremos el siguiente argumento para el mismo enunciado: Si $x = \sqrt{2}$, $y = 2\log_2 3$ entonces x^y es racional.
El último argumento también es una demostración del enunciado, con la diferencia de que es una demostración *constructiva*. En muchas aplicaciones queremos encontrar una solución real a un problema y no solamente saber que ésta existe. Por lo tanto tiene sentido considerar una aproximación constructiva de la lógica. La lógica que cumple con este requerimiento, en el contexto proposicional, es la lógica proposicional intuicionista.

En la lógica proposicional intuicionista una proposición es una especificación que *describe* un problema a resolver y la solución a un problema expresado en una proposición es una *demostración*. Si la proposición tiene una demostración, es decir si tiene solución, se dice que es verdadera y éste es el único criterio para mostrar la veracidad de alguna proposición. En el sentido constructivo, la noción de verdad no es primitiva como en la lógica clásica. Por la misma razón se entiende que una proposición es falsa si existe una refutación de ella, es decir si al asumir que hay una demostración de élla, llegamos a una contradicción. Por este motivo a la lógica proposicional intuicionista o constructiva se le considera como una lógica de información positiva, ya que se debe tener evidencia explícita, en forma de prueba, para afirmar la veracidad o falsedad de una proposición. Es esta característica la causa de que esta lógica sea básica para fundamentar conceptos computacionales. En este capítulo desarrollamos los conceptos sintácticos básicos de la lógica proposicional intuicionista mediante un sistema de deducción natural.

1.1. Lógica proposicional intuicionista

Definición 1.1. *La sintaxis de la lógica proposicional intuicionista (LPI) es la misma que la de la lógica clásica, su sintaxis en forma Backus Naur se define como:*

$$\Phi ::= Var \mid (\Phi \vee \Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \to \Phi) \mid \bot$$

$$Var ::= p \mid q \mid r \mid \ldots$$

Los conectivos $\leftrightarrow$, $\neg$, y el símbolo $\top$ son abreviaturas:

- $A \leftrightarrow B$ abrevia a $(A \to B) \wedge (B \to A)$;
- $\top$ abrevia a $\bot \to \bot$.

Usaremos la convención de que la implicación asocia hacia la derecha, es decir $A \to B \to C$ la entendemos como $A \to (B \to C)$, además asumiremos que la negación es el conectivo con la más alta precedencia y será la implicación la de menor precedencia; la conjunción y la disyunción tendrán la misma precedencia.

1.2. Semántica

En LPI la noción de veracidad de alguna proposición no es como en la lógica clásica. Para aclarar ésto recordemos brevemente la semántica para la lógica clásica donde la noción de verdad es primitiva y no depende de razonamiento o acción alguna. Toda proposición tiene un valor de verdad: verdadero (1) o bien falso (0), donde falso es lo mismo que no verdadero y no caben otras posibilidades.

Definición 1.2. *Un estado de las variables de la lógica proposicional es una función* $\mathcal{I} : \Phi \to \{0, 1\}$

Definición 1.3. *Dada una formula proposicional A se define el valor de verdad,* $\mathcal{I}(A)$*, como:*

- $\mathcal{I}(\bot) = 0$
- $\mathcal{I}(\neg B) = 1$ *si y sólo si* $\mathcal{I}(B) = 0$
- $\mathcal{I}(B \to C) = 1$ *si y sólo si* $\mathcal{I}(B) = 0$ *o* $\mathcal{I}(C) = 1$
- $\mathcal{I}(B \vee C) = 0$ *si y sólo si* $\mathcal{I}(B) = \mathcal{I}(C) = 0$
- $\mathcal{I}(B \wedge C) = 1$ *si y sólo si* $\mathcal{I}(B) = \mathcal{I}(C) = 1$

En particular se tiene que dada cualquier fórmula A, necesariamente A es verdadera o falsa, es decir $\mathcal{I}(A \vee \neg A) = 1$. Por ejemplo el siguiente enunciado:

No somos los únicos seres vivientes inteligentes en el universo

debe ser falso o verdadero en la lógica clásica, pero nótese que no hay forma de demostrar, hasta ahora, la veracidad o falsedad del enunciado. Otro ejemplo:

Los Borogroves son inteligentes o los Borogroves no son inteligentes

El valor de verdad del último enunciado es verdadero de acuerdo a la lógica clásica, ya que o se cumple que *los Borogroves son inteligentes* o bien su negación. Sin embargo, no podemos saber cual se cumple puesto que no sabemos que es un Borogrove.

En contraste, en LPI en vez de una noción predefinida de verdad se usa la noción de *demostración*, donde según Heyting una demostración no es más que un reflejo borroso de lo que una demostración es en verdad (véase [2]). Una demostración de una proposición A es una construcción matemática que establece A. Es así que un enunciado es verdadero si tenemos una demostración de él, y falso si suponer la existencia de una demostración del enunciado conduce a una contradicción; por lo que dada cualquier proposición A no se puede asegurar si A es verdadera o falsa.

1.3. La interpretación BHK

En *LPI* la interpretación de los conectivos lógicos está en términos de las nociones primitivas de *construcción* y de *demostración* , esto se conoce como la interpretación de Brouwer-Heyting-Kolmogorov (BHK).

Definición 1.4 (Interpretación BHK)**.** *La interpretación de Brouwer-Heyting-Kolmogorov se define de la siguiente manera.*

- *La demostración de una variable proposicional se supone conocida y dada por un contexto dado.*

- *Una demostración P de $A \wedge B$ es un par que consiste de una demostración de A y una demostración de B*
- *Una demostración de $A \to B$ es una función f, tal que para toda demostración P de A, $f(P)$ es una demostración de B*
- *Una demostración P de $A \vee B$ es un par (i, P) donde $i \in \{1, 2\}$ y*
  - *Si $i = 1$ entonces P es una demostración de A*
  - *Si $i = 2$ entonces P es una demostración de B*
- *No hay demostración de $\bot$*

La negación

La negación es un caso especial en LPI, como ya mencionamos de manera intuicionista $\neg A$ es simplemente $A \to \bot$; donde según la *interpretación BHK*, una construcción de $\neg A$ es una función que transforma toda demostración hipotética de A a una demostración de una contradicción. Nótese que demostrar $\neg A$ es más fuerte que probar que no existe una demostración de A, ya que suponer una demostración de A implica que existe una prueba del falso lo cual no puede suceder por lo que demostrar $\neg A$ nos dice que no puede existir una demostración de A.

Ejemplo 1.1. *Sean M y N enunciados matemáticos arbitrarios. Se puede demostrar de manera intuicionista que $M \to (N \to M)$ como sigue: sea A una demostración de M entonces la función constante f_1 que a toda demostración B de N la transforma en una demostración de M es una demostración de $N \to M$ y por lo tanto la función constante f_2 que transforma una demostración de N a una demostración de $M \to N$ es una demostración de $M \to (N \to M)$*

Con la interpretación *BHK* algunas tautologías de la lógica clásica no lo son más en LPI, por ejemplo $p \vee \neg p$, $p \in Var$, ya que habría que exhibir una demostración de p o de $\neg p$, lo cual no siempre es posible. Por ejemplo, digamos que p representa a la afirmación *la máquina de Turing T siempre se detiene*; para que sea válida en el sentido intuicionista habría que exhibir una demostración de p o de $\neg p$ pero sabemos que el problema de la detención no es decidible, por lo tanto no se puede exhibir una demostración de p ni de $\neg p$. Entonces que una fórmula sea válida de manera clásica no implica que lo sea en LPI, lo que sí es cierto es que toda fórmula válida en el sentido intuicionista es válida de manera clásica. Acontinuación enlistamos más ejemplos.

Ejemplo 1.2. *Fórmulas válidas en lógica clásica pero no en LPI.*

- $\neg\neg p \to p$
- $p \vee \neg p$
- $((p \to q) \to p) \to p$
- $(\neg p \to \neg q) \to (q \to p)$

- $\neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q)$
- $(p \to q) \to (\neg p \to q) \to q$
- $((p \leftrightarrow q) \leftrightarrow r) \leftrightarrow (p \leftrightarrow (q \leftrightarrow r))$
- $(p \to q) \leftrightarrow (\neg p \vee q)$
- $(p \vee q \to p) \vee (p \vee q \to q)$
- $(\neg\neg p \to p) \to p \vee \neg p$

Ejemplo 1.3. *Algunas fórmulas válidas en LPI y lógica clásica.*

- $\bot \to p$
- $p \to q \to p$
- $(p \to q \to r) \to (p \to q) \to p \to r$
- $p \to \neg\neg p$
- $\neg\neg\neg p \to \neg p$
- $(p \to q) \to (\neg q \to \neg p)$
- $\neg(p \vee q) \leftrightarrow (\neg p \wedge \neg q)$
- $(p \wedge q) \to r) \leftrightarrow (p \to (q \to r))$
- $\neg\neg(p \vee \neg p)$
- $(p \vee \neg p) \to \neg\neg p \to p$

La interpretación BHK proporciona una semántica intuitiva para LPI y si bien ésta puede formalizarse no es de nuestro interés hacerlo aquí por lo que remitimos al interesado a las semánticas de mundos posibles de Kripke o de álgebras de Heyting (véase [14] y [7]).

1.4. Deducción natural

En 1935 el matemático alemán Gerhard Gentzen introdujo la deducción natural. En su artículo [11] escribe: *Mi punto inicial fue este: La formalización de la deducción lógica, especialmente como ha sido desarrollada por Frege, Russel, y Hilbert es lejana de las formas de deducción usadas en la práctica en demostraciones matemáticas.* Gentzen intentaba dar un sistema formal en el cual se expresaran de una manera más *natural* las demostraciones matemáticas. El resultado fue un cálculo de deducción natural, además de un resultado importante, el cual afirma que toda demostración lógica pura puede ser simplificada y puesta en forma normal véase [11], este tema se discutirá en capítulos posteriores.

Un sistema de deducción natural consiste de reglas de inferencia para introducir y eliminar cada uno de los conectivos lógicos. Así una demostración es una sucesión de aplicaciones de estas reglas, la cual puede ser escrita en forma de árbol. Además se pueden hacer hipótesis temporales durante la demostración, las cuales se pueden descargar al incorporarlas a la conclusión.

1.4.1. Significado de conectivos

Antes de definir un sistema de deductivo para LPI es importante entender el significado de los conectivos, ya que se busca que las reglas del sistema deductivo correspondan de manera natural a éstas. La interpretación BHK nos da ya la mitad del significado de los conectivos, y corresponde a las reglas de introducción de conectivos dadas abajo. La otra mitad está relacionada a la información que podemos obtener de un conectivo dado, y corresponde a las reglas de eliminación.

- Información básica: A es cierta, lo cual denotamos como A true. Obsérvese que la noción de verdad puede ser la clásica o bien la noción constructiva dada por la interpretación BHK.

- Conjunción: $A \wedge B$ es cierta solo sí ambas A y B son ciertas. Lo cual nos lleva a la regla:

$$\frac{A\ \mathsf{true} \quad B\ \mathsf{true}}{A \wedge B\ \ \mathsf{true}}$$

  La siguiente cuestión es preguntarnos como usar la información $A \wedge B$ true, a cuya respuesta nos lleva de nuevo el razonamiento natural:

$$\frac{A \wedge B\ \ \mathsf{true}}{A\ \mathsf{true}} \qquad \frac{A \wedge B\ \ \mathsf{true}}{B\ \mathsf{true}}$$

- Implicación: ¿Cuando es verdadera una implicación? el razonamiento matemático nos dice que la implicación $A \to B$ es cierta si el suponer el antecedente A como cierto nos permite probar que el consecuente B es cierto, lo que nos lleva a la siguiente regla de introducción:

$$\frac{\begin{array}{c}[A\ \mathsf{true}] \\ \vdots \\ B\ \mathsf{true}\end{array}}{A \to B\ \mathsf{true}}$$

  Aquí los corchetes que encierran a A true, indican que en la conclusión tal hipotésis fue descargada, es decir despues de introducir la implicación tal hipótesis no existe más. Esto corresponde a una demostración hipotética de A.

La regla de eliminación de la implicación modela una forma de razonamiento conocida desde Aristóteles y llamada *modus ponens*. De las afirmaciones $A \to B$ true y A true podemos obtener la afirmación B true.

$$\frac{A \to B \text{ true} \quad A \text{ true}}{B \text{ true}}$$

- La disyunción nos lleva a las siguientes reglas de introducción

$$\frac{A \text{ true}}{A \vee B \text{ true}} \qquad \frac{B \text{ true}}{A \vee B \text{ true}}$$

Lo cual captura el hecho de que una disyunción es cierta sólo si alguna de sus dos componentes lo es. Para obtener la regla de eliminación debemos considerar como utilizar correctamente la afirmación $A \vee B$ true dado que no sabemos con certeza cual de las dos componentes es cierta. Si tratamos de probar C true a partir de $A \vee B$ true debemos hacerlo sin importar si A true o B true. Esto nos lleva a hacer una demostración por casos capturada en la siguiente regla:

$$\frac{A \vee B \text{ true} \qquad \begin{array}{c}[A \text{ true}] \\ \vdots \\ C \text{ true}\end{array} \qquad \begin{array}{c}[B \text{ true}] \\ \vdots \\ C \text{ true}\end{array}}{C \text{ true}}$$

Al igual que en la introducción de la implicación los corchetes indican que tal hipótesis ha sido descargada.

- La falsedad $\bot$ representa una contradicción y no debería ser probable, por lo que no tiene regla de introducción. Inversamente si llegamos en algún momento a la afirmación $\bot$ true deberíamos poder concluir cualquier cosa, lo cual genera la regla de eliminación:

$$\frac{\bot \text{ true}}{A \text{ true}}$$

1.4.2. El sistema DN

Para formalizar la lógica proposicional intuicionista definimos un sistema formal, llamado DN. Las reglas de este sistema expresarán de manera formal el significado de los conectivos recién discutido.

Definición 1.5. *Un contexto es un conjunto finito de fórmulas,* $\Gamma = \{A_1, A_2, ..., A_n\}$

Escribimos Γ, Δ en lugar de de $\Gamma \cup \Delta$ y a $\Gamma, \{A\}$ como Γ, A. Cuando escribimos Γ, Δ entedemos que $\Gamma \cap \Delta = \varnothing$

Definición 1.6. *Un juicio en deducción natural es una expresión de la forma* $\Gamma \vdash A$ *donde* Γ *es un contexto y* A *es una formula. En particular el juicio* $\vdash A$ *significa* $\varnothing \vdash A$*, y se lee como "A es un teorema".*

El juicio $\Gamma \vdash A$ expresa la relación de deducción o derivabilidad de la fórmula A a partir de las hipótesis en Γ. Esta relación se define a continuación.

Definición 1.7. *La relación de derivabilidad* $\Gamma \vdash A$ *se define recursivamente como sigue:*

- *Hipótesis*

$$\frac{}{\Gamma, A \vdash A}\ (Hip)$$

- *Implicación*

$$\frac{\Gamma, B \vdash A}{\Gamma \vdash B \to A}\ (\to I) \quad \frac{\Gamma \vdash B \to A \qquad \Gamma \vdash B}{\Gamma \vdash A}\ (\to E)$$

- *Conjunción*

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge I) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge E) \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge E)$$

- *Disyunción*

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee I) \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\vee I) \quad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\ (\vee E)$$

- *Falso*

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot E)$$

A continuación damos reglas estructurales de la noción de derivación las cuales son un valioso auxiliar para derivar fórmulas.

Proposición 1.1. *Se cumplen las siguientes propiedades*

- *Intercambio de premisas: Si* $\Gamma, A, B \vdash C$ *entonces* $\Gamma, B, A \vdash C$
- *Monotonía o debilitamiento: Si* $\Gamma \vdash A$ *entonces* $\Gamma, B \vdash A$.
- *Contracción: Si* $\Gamma, A, A \vdash B$ *entonces* $\Gamma, A \vdash B$
- *Sustitución: Si* $\Gamma, A \vdash B$ *y* $\Gamma \vdash A$ *entonces* $\Gamma \vdash B$

Demostración. Las demostraciones de estas propiedades son por inducción sobre $\vdash$ y se omiten. □

1.4.3. Reglas para la negación

Ya hemos dicho que en presencia de $\bot$, la negación no es un conectivo independiente, si no que se define como

$$\neg A =_{def} A \to \bot$$

Dependiendo de las reglas definidas para la negación existen tres sistemas de deducción natural:

- Minimal. No hay reglas para el $\bot$ ni para la negación. Un sistema de deducción natural para la lógica minimal puede ser el presentado anteriormente, sin la regla para el falso.
- Intuicionista. Se obtiene al agregar a la lógica minimal la siguiente regla

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}$$

- Clásica. La negación se considera como un conectivo primitivo regido por la regla del tercero excluido:

$$\frac{}{\Gamma \vdash \neg A \vee A}\,(TE)$$

  o equivalentemente a alguna de las siguientes reglas

  - $\dfrac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\,(\neg\neg E)$
  - $\dfrac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\,(RAA)$

Ejemplo 1.4. *Veamos ahora algunos ejemplos de derivaciones.*

- $\vdash \bot \to p$

$$\dfrac{\dfrac{\bot \vdash \bot}{\bot \vdash p}\,(\bot E)}{\vdash \bot \to p}\,(\to I)$$

- $\vdash p \to \neg\neg p$

  sea $\Gamma = \{p \to \bot, p\}$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash p \to \bot \qquad \Gamma \vdash p}{\Gamma \vdash \bot}\,(\to E)}{\vdash (p \to \bot) \to \bot}\,(\to I)}{\vdash p \to \neg\neg p}\,(\to I)$$

- $\vdash (p \to q) \to (\neg q \to \neg p)$
  sea $\Gamma = \{p \to q, q \to \bot, p\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash p \qquad \Gamma \vdash p \to q}{\Gamma \vdash q}\,(\to E) \qquad \Gamma \vdash q \to \bot}{\Gamma \vdash \bot}\,(\to E)}{\{p \to q, q \to \bot\} \vdash p \to \bot}\,(\to I)}{\{p \to q\} \vdash \neg q \to \neg p}\,(\to I)}{\vdash (p \to q) \to (\neg q \to \neg p)}\,(\to I)$$

- $((p \wedge q) \to r) \to (p \to (q \to r))$
  sea $\Gamma = \{p, q, (p \wedge q) \to r\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash p \qquad \Gamma \vdash q}{p \wedge q}\,(\wedge I) \qquad \Gamma \vdash (p \wedge q) \to r}{r}\,(\to E)}{\{p, (p \wedge q) \to r\} \vdash q \to r}\,(\to I)}{\{(p \wedge q) \to r\} \vdash p \to (q \to r)}\,(\to I)}{\vdash ((p \wedge q) \to r) \to (p \to (q \to r))}\,(\to I)$$

- $\vdash (\neg p \vee \neg q) \to \neg(p \wedge q)$
  sea $\Gamma = \{\neg p \vee \neg q, p \wedge q\}$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \neg p \vee \neg q \qquad \dfrac{\dfrac{\Gamma, p \to \bot \vdash p \wedge q}{\Gamma, q \to \bot \vdash q} \qquad \Gamma, q \to \bot \vdash p \to \bot}{\Gamma, p \to \bot \vdash \bot} \qquad \dfrac{\dfrac{\Gamma, q \to \bot \vdash p \wedge q}{\Gamma, q \to \bot \vdash q} \qquad \Gamma, q \to \bot \vdash q \to \bot}{\Gamma, q \to \bot \vdash \bot}}{\Gamma \vdash \bot}\,(\vee E)}{\{\neg p \vee \neg q\} \vdash (p \wedge q) \to \bot}\,(\to I)}{\vdash (\neg p \vee \neg q) \to \neg(p \wedge q)}\,(\to I)$$

Para finalizar esta sección observemos algunas características del sistema de deducción natural intuicionista.

- Logra expresar de una manera natural el razonamiento que se usa en las demostraciones matemáticas.
- Da una formalización de las demostraciones matemáticas.
- Es sistemático, a cada conectivo lógico le corresponde una regla que lo introduce y una que lo elimina; las reglas de introducción las podemos ver como definiciones de los conectivos que introducen, y a las reglas de eliminación como pautas para obtener información a partir de tales definiciones.

1.5. Normalización de demostraciones

La normalización de demostraciones se remonta a Gentzen, quién notó que las demostraciones podían contener *redundancias* o *desvíos*, y se dio cuenta que la mayoría de las dificultades en el análisis de las demostraciones eran debido a estas redundancias (véase[17]). El pensó que eliminando tales redundancias el análisis de las demostraciones podría simplificarse, lo que lo llevó a probar que toda demostración puede reducirse a una demostración equivalente que estuviera en forma normal, es decir que no tuviera redundancias. Además construyó un algoritmo para producir una demostración en forma normal ([1]). Sin embargo su marco de trabajo fue el cálculo de secuentes. Más tarde Prawitz retomó el problema de normalización de demostraciones pero en el marco de la deducción natural, logrando explicar formalmente lo que son las redundancias en los sistemas de deducción natural además de demostrar que toda demostración puede ser reducida a una forma normal, la cual es única. Más aún probó que toda secuencia de reducción termina, lo cual se conoce como *normalización fuerte*. Para mayor información sobre la normalización de demostraciones véase [4].

Definición 1.8. *Una desviación en una demostración consiste de usar la regla de eliminación de un conectivo inmediatamente después de haber usado su regla de introducción.*

Ejemplo 1.5. *Demostraciones redundantes.*
Observemos las siguiente demostración de p, donde $\Gamma = \{p, q\}$

$$\dfrac{\dfrac{\Gamma \vdash p \qquad \Gamma \vdash q}{\Gamma \vdash p \wedge q}\,(\wedge I)}{\Gamma \vdash p}\,(\wedge E)$$

La demostración es redundante, una demostración sin desvíos sería:

$$\Gamma \vdash p$$

Definición 1.9. *El proceso de eliminar demostraciones redundantes, es llamado normalización de demostraciones. Una derivación que no tiene desvíos o redundancias se dice que está en forma normal.*

Ejemplo 1.6. *Veamos otro ejemplo*

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, A \vdash A}}{\Gamma \vdash A \to A}\,(\to I) \qquad \dfrac{\vdots}{\Gamma \vdash A}}{\Gamma \vdash A}\,(\to E) \qquad \to \qquad \begin{array}{c}\vdots\\ \Gamma \vdash A\end{array}$$

La derivación de la izquierda es redundante, ya que después de introducir la implicación inmediatamente la eliminamos. Una demostración que elimina ese desvío es la derivación de la derecha.

Teorema 1.1. *Para toda demostración existe una secuencia de reducciones que conducen a una demostración en forma normal de la misma proposición*

Demostración. Se dará un esquema de la demostración solamente.

La idea es usar una medida bien fundada para las derivaciones y redundancias. De tal forma que si reducimos una redundancia de medida máxima la demostración resultante tenga una medida más pequeña. Dado que el orden de las medidas esta bien fundado si repetimos el proceso éste debe terminar en algún momento por principio del buen orden. Una medida apropiada de una derivación D es el par (d, n), ordenado lexicográficamente, donde d es el grado de la derivación D, y n es el número de redundancias en D de grado d; el grado de una derivación se define como el máximo del grado de las redundancias que ocurren en élla; el grado de una redundancia se define como el grado de la proposición que se elimina; el grado de $\perp$ es cero el grado de $A \star B$ es uno más la súma de los grados de A y B, $\star \in \{\wedge, \vee, \rightarrow\}$.

Lo crucial de la demostración consiste en demostrar que si reducimos una redundancia de grado máximo en una derivación, entonces la derivación resultante tiene medida menor estríctamente. □

Como consecuencia del teorema anterior tenemos que si existe una demostración de una proposición P entonces P tiene una demostración en forma normal.

Teorema 1.2 (Normalización fuerte)**.** *No hay una secuencia infinita de reducciones que inicien de alguna demostración. Toda secuencia de reducciones obtenida al eliminar repetidamente redundancias debe terminar en una demostración en forma forma normal.*

Demostración. La demostración está fuera del alcance del tema de este trabajo una demostración puede ser consultada en [] □

Un algoritmo para normalizar demostraciones no será dado pues también esta fuera del alcance de este trabajo, sin embargo nos interesará contestar lo siguiente sobre las redundancias en las demostraciones:

- ¿Cual es su significado?
- ¿Tienen algún propósito?

En el sistema de deducción natural para LPI tenemos una representación de las demostraciones en forma de árbol, la cual no es siempre adecuada para su manipulación, sobre todo cuando la demostración es demasiado grande, como la última derivación del ejemplo 1.4, por lo que sería recomendable tener alguna representación más *lineal*, es decir, más fácil de manipular. Es de interés tener esta representación de las demostraciones ya que en algunas ramas de la lógica, como la teoría de la demostración, la demostración es el objeto principal de estudio, por ejemplo en teoremas como el de normalización de demostraciones el manejo de demostraciones es indispensable por lo que tener una notación simple para éllas es mandatorio. A este respecto pensemos, por ejemplo, en la simplicidad de la notación numérica usual, digamos n en comparación con la notación

primitiva como sucesión de símbolos $|||\ \ldots$, con la cual sería prácticamente imposible desarrollar la Teoría de los Números.

En este capítulo se ha presentado la lógica proposicional intuicionista LPI mediante un sistema de deducción natural para LPI que formaliza el concepto matemático de demostración.

En el siguiente capítulo se dará una breve introducción a los sistemas de tipos, que nos servirá como preámbulo a la descripción del cálculo lambda, sin tipos y con tipos simples, el cual nos permitirá, en primera instancia, definir una notación lineal para las demostraciones en el sistema de deducción natural.

Capítulo 2

Sistemas de tipos

Un tipo en un lenguaje de programación puede ser visto como el rango de valores posibles que una expresión puede tomar. Por ejemplo una variable x de tipo booleano puede y debe tomar únicamente valores booleanos durante la ejecución del programa. Los lenguajes de programación donde las expresiones reciben un tipo se llaman lenguajes tipados, en contraste con aquellos donde no hay tal restricción, es decir aquellos que no tienen tipos, o equivalentemente, en los que todos los programas pertenecen a un tipo universal. En estos lenguajes las operaciones podrían aplicarse a argumentos inapropiados, por ejemplo, `sucesor true`, y el resultado podría ser un valor fijo arbitrario de falla, una excepción particular o bien un error de ejecución no capturado.
Un *sistema de tipos* es una componente de un lenguaje tipado que monitorea los tipos de las variables y expresiones de un programa. Los sistemas de tipos se usan para determinar si los programas se comportan correctamente. Sólo los programas que son aceptados por el sistema de tipos deben ser considerados como válidos y susceptibles de evaluación, si hay un error de tipos el programa debe ser deshechado antes de intentar su ejecución.
Un lenguaje es tipado en virtud de la existencia de un sistema de tipos para él, independientemente de que los tipos figuren o no explícitamente en el código. Un lenguaje tipado es *explícitamente tipado* si los tipos forman parte de su sintaxis e *implícitamente tipado* en otro caso. No existe un lenguaje en uso común que sea implícitamente tipado estrictamente hablando aunque en lenguajes como ML y Haskell es posible escribir grandes fragmentos de código omitiendo la información de tipos, en cuyo caso los sistemas de tipos asignan tipos de manera automática a dichos fragmentos.
Los lenguajes de programación con tipos tienen las siguientes ventajas:

- *Economía en la ejecución.* La información que dan los tipos permite llevar acabo en tiempo de ejecución la aplicación de las operaciones adecuadas sin la necesidad de pruebas adicionales que resultan costosas.
- *Economía en compilación.* La información que proporcionan los tipos puede ser organizada en interfaces o módulos, permitiendo compilar de

manera independiente cada uno de los módulos. En la compilación de grandes sistemas esto provoca un ahorro ya que si un módulo cambia no provoca que otros módulos sean recompilados.

- *Son más fáciles de implementar eficientemente.*

Las siguientes son características importantes de los sistemas de tipos:

- Permiten descubrir errores de programación tempranamente.
- Los tipos documentan un programa de manera más simple y manejable que los comentarios.
- Deben tener un algoritmo que asegure que un programa es bien comportado, y prevenir errores antes de que éstos sucedan.
- Deben ser transparentes. El programador debe ser capaz de predecir *fácilmente* si un programa tiene el tipo correcto, en caso de que no, la razón debe ser *evidente*.
- Las declaraciones de tipos deben ser verificados estáticamente en la medida de lo posible, en donde no se pueda la verificación debe ser dinámica.
- Soportan abstracción, lo que permite la descripción de interfaces.
- Son seguros, es decir los programas correctamente tipados no pueden funcionar mal.

No es nuestra intención discutir más las ventajas o desventajas de los lenguajes tipados, el lector interesado en ahondar en este tema puede consultar [21].

En nuestro caso no estamos interesados en un lenguaje de programación particular sino en prototipos de éstos. En este capítulo presentamos un prototipo de lenguaje de programación llamado cálculo lambda, en particular nos enfocaremos en su relación con la lógica y en los beneficios prácticos que resultan de la misma, principalmente en los lenguajes de programación funcional.

Iniciamos con el cálculo lambda puro, un caso extremo de lenguaje sin tipos donde no hay error posible: la única operación es la aplicación de funciones y dado que todas las expresiones son funciones esta operación nunca falla.

2.1. Cálculo lambda puro

Leibniz buscaba un lenguaje *universal* en el que pudiera expresar *todos* los problemas (véase [15]), para después intentar resolverlos mediante algún método. Suponiendo que existiera tal lenguaje ¿se podrían resolver todos los problemas expresados en este lenguaje? fue una de las preguntas que planteó. Pero ¿como se podría probar *formalmente* que sí o que no?. Esta pregunta fue formalizada tiempo después y es lo que se conoce como el *Entscheidungsproblem*

o problema de la decisión. En 1936 el *Entscheidungsproblem* fue resuelto por Alonzo Church y Alan Turing (véase [15]), de manera independiente, cada uno usó un método distinto, pero con la misma idea: formalizar la noción de *decidible*. Church lo hizo mediante el cálculo lambda, mientras que Turing con la definición de la clase de máquinas que hoy llevan su nombre. La conclusión a la que ambos llegaron fue negativa. Alan Turing demostró la equivalencia de ambos modelos tiempo después.

Alonzo Church presentó, en 1930, el cálculo lambda como un conjunto de postulados con el propósito de fundamentar a la lógica (para mayor referencia consúltese [25]), propósito fiel a la escuela formalista de Hilbert, quienes pretendían construir un sistema formal para fundamentar a las matemáticas (véase [12]). Sin embargo, Kleene y Rosser demostraron que el cálculo lambda era un sistema inconsistente (véase [23]), por lo que el propósito original del cálculo lambda no se cumplió. Sin embargo, Church declararía que podría tener usos distintos en el futuro (véase [25]). Fueron casi proféticas las palabras de Church, ya que Kleene demostraría que el cálculo lambda era un sistema de computo universal (véase [22]); más tarde John McCarthy en 1950 sería inspirado por el cálculo lambda para crear LISP (véase [22]) (aunque McCarthy no entendía mucho del cálculo lambda, ya que en su artículo *Recursive Functions of symbolic expressions and their Computation, Comunications of the ACM.* escribió que la notación lambda era inadecuada para expresar funciones recursivas véase [18], lo cual es falso). A principios de los sesentas Peter Landin demostró que la semántica de los lenguajes de programación imperativos podía ser descrita con el cálculo lambda (véase [22]), lo cual introdujo la principal notación de los lenguajes de programación funcionales e influyó en el diseño de lenguajes tanto imperativos como funcionales. Esto guió a Christopher Strachey a realizar trabajos importantes en el fundamento de la semántica denotacional (véase [22]); el trabajo de Strachey inspiró a Dana Scott a inventar la teoría de dominios (véase [22]), la cual es una de las más importantes áreas en ciencia de la computación teórica. En resumen el cálculo lambda ha tenido gran importancia en áreas como

- Diseño de lenguajes de programación.
- Semántica de lenguajes de programación.
- Arquitectura de computadoras
- El estudio de preguntas fundamentales en computación

Para más información sobre el impacto del cálculo lambda en las ciencias de la computación; véase [13]

2.1.1. Sintaxis del cálculo lambda

La sintaxis del cálculo lambda es sencilla, los términos básicos son variables las cuales representan funciones. Además tenemos dos operaciones básicas:

- *Abstracción*: permite definir funciones sin la necesidad de nombrarlas, es decir, funciones anónimas. Así el término $\lambda x.e$ abstrae la variable x en la expresión e, es decir esta expresión define anónimamente a la función que a toda x le asigna la expresión e. El punto denota el ligado de x en e, además indica que el alcance del ligado para x se extiende tanto como sea posible. Por ejemplo $\lambda x.xy$ significa $\lambda x.(xy)$ y no $(\lambda x.x)y$
- *Aplicación*: denota a la acción de pasar un argumento dado a una función. Sintácticamente pueden ser vistas como una concatenación de términos lambda, que asocia hacia la izquierda de manera que $e_1\ e_2\ e_3$ significa $(e_1\ e_2)\ e_3$.

Veamos unos ejemplos.

Ejemplo 2.1. *Funciones anónimas.*

- $\lambda x.x$ *denota a la función identidad* $x \mapsto x$
- $\lambda x.x + 1$ *denota a la función sucesor* $x \mapsto x + 1$
- $\lambda x.x^2$ *denota a la funcion* $x \mapsto x^2$

Ejemplo 2.2. *Aplicación de funciones.*

- $(\lambda x.x)\ s\ x$
- $(\lambda y.y + 1)\ true$
- $(\lambda x.x\ x)\ (\lambda y.y\ y)\ 1$

Ya dijimos que una aplicación denota la acción de pasar un argumento a una función, ¿pero cuales son las reglas que permiten evaluar una aplicación?, ya que ésta sólo es una representacíon sintáctica. De manera intuitiva el término $(\lambda x.x{+}1)2$ debería evaluarse a 2+1, y el término $(\lambda x.xy)(\lambda w.w)$ a $(\lambda w.w)y$ y éste finalmente a y, de donde podemos deducir que la evaluación en el cálculo lambda es similar a una simplificación o reducción. La regla que nos permitirá evaluar funciones es la β-reducción, cuya definición formal daremos más adelante.

Definición 2.1 (Lambda términos (Λ))**.** *La clase de lambda términos, denotada con* Λ*, es la menor clase que satisface lo siguiente:*

- *Si* x *es una variable entonces* $x \in \Lambda$
- *Si* $M \in \Lambda$ *entonces* $\lambda x.M \in \Lambda$
- *Si* $M, N \in \Lambda$ *entonces* $MN \in \Lambda$

Observación 2.1. *Las expresiones que usan los símbolos* $+, *, -, 1, 2, 3, ...n$ *en los ejemplos anteriores están dadas de manera informal, ya que éstos símbolos no forman parte del los lambda términos. Más adelante se dará de manera formal la definición de algunas de estas expresiones.*

Para facilitar la escritura de abstracciones usaremos la siguiente convención

$$\lambda x_1 x_2 ... x_n.e =_{def} \lambda x_1 \lambda x_2 ... \lambda x_n.e$$

Dado que la abstracción $\lambda x.e$ define un ligado de x en e, surgen los conceptos de variable ligada y libre similares a los de la lógica de predicados.

Definición 2.2 (Variables libres)**.** *El conjunto de variables libres de $M \in \Lambda$, denotado $VL(M)$, se define como:*

- $VL(x) = \{x\}$
- $VL(MN) = VL(M) \cup VL(N)$
- $VL(\lambda x.M) = VL(M) - \{x\}$

Ejemplo 2.3. *Variables libres.*

- $VL(\lambda x.xy) = \{y\}$
- $VL(\lambda x.(\lambda y.xy)) = \varnothing$
- $VL(x(\lambda z.zy)) = \{x, y\}$

Definición 2.3 (Variables ligadas o acotadas)**.** *El conjunto de variables ligadas de $M \in \Lambda$, VA(M), se define como:*

- $VA(x) = \varnothing$
- $VA(MN) = VA(M) \cup VA(N)$
- $VA(\lambda x.M) = VA(M) \cup \{x\}$

Ejemplo 2.4. *Variables ligadas.*

- $VA(\lambda x.(\lambda y.x + y)) = \{x, y\}$
- $VA(xyz) = \varnothing$
- $VA(\lambda x.xz) = \{x\}$

2.1.2. α-equivalencia

Obsérvese que los lambda términos $\lambda x.x$ y $\lambda y.y$ representan a la función identidad, ya que sólo difieren en el nombre de la variable ligada, de manera que para el proceso de evaluación nos gustaría tratarlos como iguales. En estos casos decimos que estos lambda términos son $\alpha-$equivalentes.

Definición 2.4. *Decimos que dos expresiones e, e' son α-equivalentes y escribimos $e \equiv_\alpha e'$ si difieren únicamente en los nombres de las variables ligadas*

En adelante consideramos a dos lambda términos α-equivalentes como iguales.

Ejemplo 2.5. *Expresiones α-equivalentes.*

- $\lambda x.xy \equiv_\alpha \lambda z.zy$
- $\lambda xyz.xyzw \equiv_\alpha \lambda mln.mlnw$

2.1.3. Substitución

Otro concepto importante en el cálculo lambda es la substitución, la cual consiste en reemplazar las presencias libres de una variable x en una expresión e, por otra expresión e'. Denotaremos esta operación como $e[x := e']$. La definición de substitución debe darse con mucho cuidado para evitar ligar variables libres, por ejemplo, considere los siguientes lambda términos: $\lambda y.y * x \equiv_\alpha \lambda z.z * x$. La substitución $[x := 2 * y]$ (la y aquí está libre), en ambos términos daría como resultado:

- $(\lambda y.y * x)[x := 2 * y] = (\lambda y.y * (2 * y))$
- $(\lambda z.z * x)[x := 2 * y] = (\lambda z.z * (2 * y))$

Nótese que las presencias de la variable y en el primer término, después de la substitución, están todas ligadas, mientras que en el segundo término no. Si evaluamos ambos términos resultantes con 3 obtenemos:

- $(\lambda y.y * (2 * y))3 \rightarrow 3 * (2 * 3)$
- $(\lambda z.z * (2 * y))3 \rightarrow 3 * (2 * y)$

Por lo tanto estas expresiones han dejado de ser α-equivalentes, lo cual es inadmisible.

Definición 2.5. *Para $N, M \in \Lambda$ y x una variable, la sustitución de x por N en M, escrito $M[x := N]$,se define como:*

- $x[x := N] = N$
- $y[x := N] = y$
- $AB[x := N] = A[x := N]B[x := N]$
- $(\lambda y.A)[x := N] = \lambda y.(A[x := N])$ *si y sólo si* $y \notin \{x\} \cup VL(N)$

La definición de substitución evita la captura de variables libres, sin embargo está definida de manera parcial ya que $(\lambda x.x+y)[y := 2*x]$ no esta definida pues $x \in VL(2 * x)$. Este problema se soluciona al trabajar con la α-equivalencia. De esta manera el problema anterior se resuelve como sigue:

$$(\lambda x.x + y)[y := 2 * x] \equiv_\alpha (\lambda w.w + y)[y := 2 * x] = \lambda w.w + (2 * x)$$

Ejemplo 2.6. *Substitución.*

- $(x + y)[x := 2] = 2 + y$
- $(\lambda z.z + x)(\lambda x.x)[x := z] = (\lambda w.w + z)(\lambda x.x)$

2.1.4. β-reducción

La regla de reducción conocida como β-reducción proporciona la semántica operacional del cálculo lambda y consiste en definir el proceso evaluación. Para esto se crea una copia del cuerpo de la lambda abstracción y las ocurrencias de las variables ligadas son remplazadas por el argumento, por ejemplo:

$$(\lambda x.x + 1)4 \rightarrow 4 + 1$$

A continuación definimos formalmente esta relación.

Definición 2.6 (β-reducción)**.** *La $\beta-$reducción de un solo paso se define como la más pequeña relación $\rightarrow_\beta$ sobre los términos lambda que satisface las siguientes reglas de inferencia:*

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

$$\frac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N} \qquad \frac{M \rightarrow_\beta M'}{NM \rightarrow_\beta NM'} \qquad \frac{M \rightarrow_\beta M'}{\lambda x.M \rightarrow_\beta \lambda x.M'}$$

A un término lambda de la forma $(\lambda x.M)N$, se le llama *β-redex*, al término $M[x := N]$ se le llama *reducto*, un término M está en β-forma normal, si no hay un término N tal que $M \rightarrow_\beta N$ Escribimos $M \rightarrow_\beta^* M'$ si M se reduce a M' en cero o más pasos. $\rightarrow_\beta^*$ es la cerradura reflexiva y transitiva de $\rightarrow_\beta$

Definición 2.7. *La relación $=_\beta$ ($\beta - equivalencia$), es la cerradura reflexiva, simétrica y transitiva de $\rightarrow_\beta$ que satisface:*

$$\frac{}{M =_\beta M}$$

$$\frac{M \rightarrow_\beta N}{M =_\beta N} \qquad \frac{M =_\beta N \; N =_\beta P}{M =_\beta P}$$

Ejemplo 2.7. *β-reducción*

- $(\lambda x.xy)(\lambda z.z) \rightarrow_\beta (\lambda z.z)y \rightarrow_\beta y$
- $(\lambda x.x)(\lambda y.z) \rightarrow_\beta (\lambda y.z)$

2.1.5. No terminación del cálculo lambda

El proceso de reducción en el cálculo lambda puede no terminar al existir expresiones que no cuentan con una forma normal. Veamos algunos ejemplos.

Ejemplo 2.8. *No términación del cálculo lambda.*

Considérese la siguiente secuencia de reducción:

$$(\lambda x.xx)(\lambda y.yyy) \rightarrow_\beta (\lambda y.yyy)(\lambda y.yyy) \rightarrow_\beta (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy) \rightarrow_\beta \ldots$$

Por lo tanto el término $(\lambda x.xx)(\lambda y.yyy)$ no tiene forma normal ya que el proceso de reducción continuará indefinidamente.

Un lambda término puede evaluarse a sí mismo, por ejemplo si definimos: $\omega = (\lambda x.xx)$, $\Omega = \omega\omega$ entonces:

$$\Omega = (\lambda x.xx)\omega \rightarrow_\beta \omega\omega = \Omega$$

De manera que Ω se reduce a sí mismo en un paso lo cual genera la sucesión infinita de reducciones

$$\Omega \rightarrow_\beta \Omega \rightarrow_\beta \Omega \rightarrow_\beta \ldots$$

2.2. Expresividad del cálculo lambda

El cálculo lambda parece ser un lenguaje demasiado primitivo y poco expresivo, sin embargo no es así puesto que puede ser utilizado para codificar estructuras como los números naturales, booleanos, pares o listas, las cuales son tipos de datos presentes en la mayoría de los lenguajes de programación modernos. De esta manera el cálculo lambda puro puede ser visto como un prototipo de lenguaje de programación funcional. Veamos a continuación algunos ejemplos de codificación.

2.2.1. Booleanos

Con el cálculo lambda podemos codificar a las constantes de la lógica (true, false), así como a las operaciones lógicas usuales como sigue:

Definición 2.8. *Funciones booleanas.*

- true $:= \lambda xy.x$
- false $:= \lambda xy.y$
- not $:= \lambda y.y$ false true
- and $:= \lambda xy.xy$ false
- if-then-else$= \lambda bac.bac$

La constante true se define como una función que recibe dos parámetros y devuelve siempre el primero; la constante false por otro lado es una función que recibe dos parámetros y devuelve siempre el segundo; la negación se define como la función que recibe una función a la cual se le pasan como parámetros

true y false; if-then-else es la conocida estructura presente en la mayoría de los lenguajes de programación; and es el operador de la lógica. Evaluemos unas de esta funciones para corroborar que la definición funciona.

Ejemplo 2.9. *Reducción de la función not.*

not true $\rightarrow (\lambda y.y$ false true) true $\rightarrow$ true false true $\rightarrow$ false

not false $\rightarrow (\lambda y.y$ false true) false $\rightarrow$ false false true $\rightarrow$ true

2.2.2. Pares

El tipo de dato producto puede modelarse como sigue:

- pair := λfsg.gfs
- fst := $\lambda x.x$ true
- snd := $\lambda x.x$ false

Ejemplo 2.10. *Si definimos* $\langle a, b \rangle$ = pair a b *entonces tenemos las siguientes reducciones:*

fst $\langle a, b \rangle \rightarrow (\lambda x.x$ true) $(\lambda g.gab) \rightarrow (\lambda g.gab)$ true $\rightarrow$ true $a\, b \rightarrow a$

snd $\langle a, b \rangle \rightarrow (\lambda x.x$ false) $(\lambda g.gab) \rightarrow (\lambda g.gab)$ false $\rightarrow$ false $a\, b \rightarrow b$

2.2.3. Números naturales

Cómo se mencionó al principio de esta sección, los números naturales no forman parte del lenguaje del cálculo lambda. Sin embargo estos se pueden definir de la siguiente manera.

Definición 2.9 (Numerales de Church)**.** *Sean s y x variables. El numeral de Church $\bar{n}$ se define como $\bar{n} = \lambda sx.s^n x$. Donde $s^n x$ significa aplicar n veces s a x.*

Ejemplo 2.11. *Algunos numerales de Church*

- $\bar{0} = \lambda sx.x$
- $\bar{1} = \lambda sx.sx$
- $\bar{2} = \lambda sx.ssx$
- $\bar{3} = \lambda sx.sssx$

Ejemplo 2.12. *Una ves definidos los números naturales, podemos definir funciones sobre éllos. Algunos ejemplos son:*

- succ= $\lambda nfx.f(nfx)$
- add= $\lambda nmfx.nf(mfx)$

- $\mathsf{mult} = \lambda mn.m(\mathsf{add}\ n)\ \bar{0}$

Acerca de como surgen estas definiciones podemos mencionar que sus orígenes están en ciertas definiciones dadas mediante especificaciones lógicas. Para mayor detalle véase [7].

2.2.4. Teorema de Church-Rosser

Considérese el siguiente término:

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

Donde e representa un término del cálculo lambda en forma normal. Tenemos las siguientes formas de evaluarlo:

1. $(\lambda x.(\lambda y.y)x)e \rightarrow_\beta (\lambda y.y)e \rightarrow_\beta e$
2. $(\lambda x.(\lambda y.y)x)e \rightarrow_\beta (\lambda x.x)e \rightarrow_\beta e$

Nótese que la forma en que se evaluó fue distinta; En la primera substituimos la x con e, en la segunda evaluamos primero la abstracción anidada sustituyendo la y por x; tendríamos entonces que $P = (\lambda y.y)e$, y $N = (\lambda x.x)e$, nótese además que al evaluar P y N obtenemos e. Este ejemplo nos permite presentar un teorema importante del cálculo lambda.

Teorema 2.1 (Church-Rosser)**.** *Sean $M, N, P \in \Lambda$ tal que $M \rightarrow_\beta^* P$ y $M \rightarrow_\beta^* N$, entonces existe $Z \in \Lambda$ tal que $P \rightarrow_\beta^* Z$ y $N \rightarrow_\beta^* Z$*

Demostración. Para una demostración de este teorema véase [23] □

Este teorema establece que si evaluamos un término M de dos manera distintas, de tal forma que los resultados parciales que obtengamos son dos términos N y P (no necesariamente en forma normal), entonces existe una serie de evaluaciones a partir de cada uno de ellos que nos conducirán al mismo resultado. Este teorema tiene mayor relevancia para los términos con forma normal pues establece que ésta es única.

Hasta aquí nuestro estudio del cálculo lambda puro, el lector interesado en profundizar en el tema puede consultar [14]. A continuación presentamos la versión con tipos simples del cálculo lambda, que corresponde a un prototipo de lenguajes de programación con tipos.

2.3. Cálculo lambda con tipos simples

Como se mencionó al principio de este capítulo, el cálculo lambda que Church presentó resultó inconsistente, sin embargo, estas inconsistencias pueden ser *reparadas* de dos maneras: reduciendo la expresividad del lenguaje o, siguiendo la idea de Russell, introduciendo *tipos* (véase [12]). En esta sección se verá el

segundo enfoque, es decir el cálculo lambda con tipos, el cual resulta ser interesante por sus propiedades, aplicaciones y distintos sabores.

En el cálculo lambda sin tipos representabamos funciones sin decir quienes eran sus dominios y codominios (de hecho ambos eran los lambda términos), lo cual podía llevarnos a situaciones no deseadas, como pasar un argumento a una función que no sepa como manejarlo, por ejemplo pasar a la función x como argumento a ella misma, es decir xx. Al asignar tipos a los lambda términos evitaremos estos problemas.

Hay dos formas de agregar tipos al cálculo lambda, *a la Curry* y *a la Church*, introducidas por Curry y Church respectivamente. Ambos estilos representan dos familias de sistemas de tipos distintas aunque equivalentes. En el cálculo lambda con tipos *a la Curry* cada término puede recibir un conjunto de tipos posibles, inclusive el vacío; otra característica importante es que los términos tienen tipos implícitos. En la versión *a la Church* los términos tienen anotaciones de tipos explícitos, de tal manera que cada uno recibe un único tipo, el cual es recuperable al analizarlo sintácticamente. Las versiones de Curry y Church del cálculo lambda corresponden a dos estilos distintos para el diseño e implementación de un lenguaje de programación, en el primero un programa puede ser escrito sin asignarle un tipo explícitamente, por lo que el compilador se encargará de verificar si se puede asignar un tipo al programa, ésto sucederá si el programa es correcto; ML y HASKELL siguen en gran medida este paradigma de programación. En la versión *a la Church*, un programa debe ser escrito junto con su tipo, en este tipo de lenguajes la verificación de tipos es más fácil, ya que los tipos no deben ser inferidos, PASCAL sigue este estilo de programación.

2.3.1. Cálculo lambda con tipos *a la Curry*

El modeo de tipificación implícita fue introducido originalmente para la teoría de combinadores (véase [14]), tiempo después la teoría fue modificada de manera natural para el cálculo lambda asignando elementos de un conjunto de tipos a los lambda térmnos sin tipos. A continuación presentamos el cálculo lambda con tipos simples *a la Curry*.

Definición 2.10. *Sea U un conjunto numerable, cuyos miembros serán llamados tipos básicos. El conjunto $\mathbb{T}$ de tipos simples es el conjunto definido por la gramática*

$$\mathbb{T} ::= U \mid \mathbb{T} \to \mathbb{T}$$

$A \to B$ es el tipo de las funciones de A en B. El constructor de tipos $\to$ se asocia hacia la derecha, es decir si $A_1, A_2, \ldots A_n \in \mathbb{T}$ entonces $A_1 \to A_2 \to \ldots \to A_n$ se entenderá como $(A_1 \to (A_2 \to \ldots (A_{n-1} \to A_n)))$.

Ejemplo 2.13. *Si el conjunto de tipos básicos es $U = \{\mathsf{Nat}, \mathsf{Bool}\}$ entonces algunos tipos son:*

$\mathsf{Nat}\to\mathsf{Nat}$. *El tipo de las funciones que a cada natural le asignan otro natural.*

$\mathsf{Bool}\to\mathsf{Bool}$. *El tipo de las funciones que a cada booleano le asignan otro booleano.*

*(*Nat→Nat*)→* Nat*. El tipo de funciones que a cada función de naturales en naturales le asignan un natural.*

*(*Nat→Nat*) → (*Bool→Bool*). El tipo de funciones que a cada función de naturales en naturales le asignan una función de booleanos en booleanos.*

Definición 2.11. *Un contexto es un conjunto de pares de la forma* $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$*. Donde* $A_1, A_2, \ldots, A_n \in \mathbb{T}$*,* $x_1, x_2, x_3, ..., \in Var$ *con* $x_i \neq x_j$ *si* $i \neq j$

Si Γ y Δ son dos contextos, escribimos Γ, Δ en lugar de de $\Gamma \cup \Delta$ y a $\Gamma, \{x : A\}$ como $\Gamma, x : A$. Cuando escribimos Γ, Δ entendemos $\Gamma \cap \Delta = \varnothing$.Un par $e : A \in \Gamma$ se lee como "la expresión e es de tipo A".

Definición 2.12. *La relación de tipado* $\Gamma \vdash e : A$ *donde* Γ *es un contexto,* e *es un lambda término y* A *es un tipo, se define recursivamente mediante las siguientes reglas de inferencia:*

$$\frac{}{\Gamma, x : A \vdash x : A}\ \textit{(Var)}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\ \textit{(I-Abs)} \qquad \frac{\Gamma \vdash r : A \to B \quad \Gamma \vdash t : A}{\Gamma \vdash rt : B}\ \textit{(E-Abs)}$$

Veamos unos ejemplos de derivaciones de tipos.

Ejemplo 2.14. *Derivaciones.*

- $\Gamma = \{t : A \to B, x : A\}$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash tx : B}\,(E - Abs)}{x : A \vdash \lambda t.tx : (A \to B) \to B}\,(I - Abs)}{\vdash \lambda xt.tx : A \to (A \to B) \to B}\,(I - Abs)$$

- $\Gamma = \{x : A, t : A \to B, r : B \to C\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash tx : B}\,(E - Abs) \qquad \Gamma \vdash r : B \to C}{\Gamma \vdash r(tx) : C}\,(E - Abs)}{x_2 : A \to B, x_3 : B \to C \vdash \lambda x.r(tx) : A \to C\}}\,(I - Abs)}{x_2 : A \to B \vdash \lambda rx.r(tx) : (B \to C) \to (A \to C)}\,(I - Abs)}{\vdash \lambda trx.r(tx) : (A \to B) \to (B \to C) \to (A \to C)}\,(I - Abs)$$

Propiedades

A continuaicón presentamos algunas de las principales propiedades del cálculo lambda con tipos simples *a la Curry*.

Definición 2.13. *El dominio de un contexto* $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$ *se define como:* $dom(\Gamma) = \{x_1, x_2, ..., x_n\}$

Esta definición será útil para enunciar algunas propiedades del cálculo lambda con tipos simples *a la Curry*.

Lema 2.1. *Supongamos que* $\Gamma \vdash t : A$ *entonces:*

1. *Si* $\Gamma \subseteq \Gamma'$ *entonces* $\Gamma' \vdash t : A$ *(Monotonía).*
2. $VL(t) \subseteq dom(\Gamma)$
3. $\Gamma' \vdash t : A$ *donde* $dom(\Gamma') = VL(t)$ *y* $\Gamma' \subseteq \Gamma$

Demostración. .

1. Inducción sobre la derivación $\Gamma \vdash t : A$. Se tienen tres casos, que son algunas de las tres reglas de derivación.
2. De manera análoga.
3. De manera análoga.

□

La siguiente propiedad analiza la forma del tipo asignado a cada forma de término. Es útil para mostrar que ciertos términos no tienen tipos.

Lema 2.2. *Lema de Generación.*

1. *Si* $\Gamma \vdash x : A$ *entonces que* $x : A \in \Gamma$.
2. *Si* $\Gamma \vdash rt : B$ *entonces existe una* A *tal que* $\Gamma \vdash r : A \to B$ *y* $\Gamma \vdash t : A$.
3. *Si* $\Gamma \vdash \lambda x.t : C$ *entonces existen* A *y* B *tal que* $\Gamma, x : A \vdash t : B$. *con* $C = A \to B$.

Demostración. Inducción sobre la derivación $\Gamma \vdash t : A$. □

Lema 2.3. *Lema de substitución.*

1. *Si* $\Gamma \vdash t : A$ *entonces* $\Gamma[B := C] \vdash t : A[B := C]$
2. *Si* $\Gamma, x : A \vdash t : B$ *y* $\Gamma \vdash r : A$ *entonces* $\Gamma \vdash t[x := r] : B$

Donde si $\Gamma = \{x_1 : A_1 \ldots x_n : A_n\}$ *entonces* $\Gamma[B := C] = \{x_1 : A_1[B := C], \ldots, x_n : A_n[B := C]\}$

Demostración. .

1. Por inducción sobre la derivación de $t : A$
2. Por inducción sobre la derivación de $\Gamma, x : A \vdash t : B$

□

Algunos autores denominan enunciado a la declaración $e : A$, donde e es el sujeto y A el predicado.

Teorema 2.2 (Reducción del sujeto o preservación de tipos)**.** *Si* $\Gamma \vdash r : A$ *y* $r \to_\beta s$ *entonces* $\Gamma \vdash s : A$

Demostración. Inducción sobre la generación de $\to_\beta$ usando 2.2 y 2.3. Esta demostración es complicada ya que no tenemos anotaciones de tipos. □

2.3.2. Cálculo lambda con tipos *a la Church*

Antes de dar la definición formal explicamos de manera superficial la diferencia entre asignar tipos *a la Curry* y *a la Church* , la cual se observa en los siguientes términos:

- $\vdash_{Curry} (\lambda x.x) : A \to A$
- $\vdash_{Church} (\lambda x : A.x) : A \to A$

En el término $\lambda x.x$, el parámetro x esta etiquetado en el sistema *a la Church* por el tipo A. El significado intuitivo es que el término $\lambda x.x$ toma el argumento x de A. Esta mención explícita del tipo en un término hace posible decidir si un término tiene cierto tipo. Para algunos sistemas *a la Curry* este problema es indecidible.

Definición 2.14. *Sea* $\mathbb{T}$ *un conjunto de tipos. El conjunto de términos se define por la siguiente gramática:*

$$\Lambda_{\mathbb{T}} ::= Var \mid \lambda x : \mathbb{T}.\Lambda_{\mathbb{T}} \mid \Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}}$$

Donde Var *denota al conjunto de variables de términos.*

Se adopta la misma convención que en el Sistema *a la Curry*, respecto al constructor $\to$, de que asocia a la derecha

Definimos de manera análoga el conjunto de tipos $\mathbb{T}$, un contexto y su dominio, a los definidos en el sitema *a la Curry* y mantenemos las mismas convenciones sobre ellos.

Definición 2.15. *La relación de tipado* $\Gamma \vdash e : A$ *donde* Γ *es un contexto,* e *es un lambda término y* A *es un tipo, se define recursivamente mediante las siguientes reglas de inferencia:*

$$\frac{}{\Gamma, x : A \vdash x : A}\ (Var)$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B}\ (I\text{-}Abs) \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash r : A}{\Gamma \vdash tr : B}\ (E\text{-}Abs)$$

Ejemplo 2.15. *Veamos las derivaciones análogas a las del ejemplo 2.14*

- $\Gamma = \{r : A \to B, x : A\}$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash rx : B}\ (E - Abs)}{x : A \vdash \lambda r : A \to B.rx : (A \to B) \to B}\ (I - Abs)}{\vdash \lambda x : Ar : A \to B.rx : A \to (A \to B) \to B}\ (I - Abs)$$

- $\Gamma = \{x : A, t : A \to B, r : B \to C\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash tx : B}\ (E - Abs) \qquad \Gamma \vdash r : B \to C}{\Gamma \vdash r(tx) : C}\ (E - Abs)}{x_2 : A \to B, x_3 : B \to C \vdash \lambda x : A.r(tx) : A \to C\}}\ (I - Abs)}{x_2 : A \to B \vdash \lambda r : (B \to C)x : A.r(tx) : (B \to C) \to (A \to C)}\ (I - Abs)}{\vdash \lambda t : (A \to B)r : (B \to C)x : A.r(tx) : (A \to B) \to (B \to C) \to (A \to C)}\ (I - Abs)$$

Observación 2.2. *Al igual que en el cálculo lambda puro, podemos definir: variables libres, variables acotadas, substitución, y β-reducción. Las definiciones son análogas y por lo tanto se omiten.*

Definición 2.16. *El conjunto de de lambda términos válidos en el sistema a la Church es*

$$\Lambda = \{t \in \Lambda_{\mathbb{T}} \mid \exists \Gamma,\ A,\ \Gamma \vdash t : A\}$$

Propiedades

En el sistema *a la Church* también se cumplen el lema de generación, substitución y la preservación de tipos. Las demostraciones son análogas a las presentadas en el sistema *a la Curry* por lo cual se omiten.

Lema 2.4. *Supongamos que $\Gamma \vdash M : A$ entonces:*

1. *si $\Gamma \subseteq \Gamma'$ entonces $\Gamma' \vdash M : A$*
2. *$VL(M) \subseteq dom(\Gamma)$*
3. *$\Gamma' \vdash M : A$ donde $dom(\Gamma') = VL(M)$ y $\Gamma \subseteq \Gamma'$*

Demostración. Análoga al caso *a la Curry*. □

La siguiente propiedad no se cumple para el cálculo lambda con tipos *a la Curry*.

Teorema 2.3 (Unicidad de tipos)**.** .

1. *Si* $\Gamma \vdash t : A$, $\Gamma \vdash t : B$ *entonces* $A = B$
2. *Si* $\Gamma \vdash t : A$, $\Gamma \vdash r : B$ *y* $t =_\beta r$ *entonces* $A = B$

Demostración. .

1. Inducción sobre la estructura de t
2. Por el teorema de *Church-Rosser* para $\Lambda_{\mathbb{T}}$ y por la propiedad de reducción del sujeto.

□

2.3.3. Problemas de decisión

Del análisis de la terna $\Gamma \vdash t : A$, surgen varios problemas de decisión, por ejemplo:

- Problema de verificación de tipos. Dados t, Γ, y $A \in \mathbb{T}$ decidir si $\Gamma \vdash t : A$.
- Problema de asignación de tipos. Dado t, decidir si existen Γ y A tal que $\Gamma \vdash t : A$.
- Problema de pertenencia o habitación (*type inhabitation*). Dado $A \in \mathbb{T}$ decidir si existen Γ y t tal que $\Gamma \vdash t : A$.

Estos no son lo únicos problemas que surgen del análisis de la terna $\Gamma \vdash t : A$, pero son los que más nos interesarán en este trabajo, ya que su importancia no solo radica en la lógica si no también en los lenguajes de programación; además la mayoría de los otros problemas resultan ser equivalentes en complejidad, en cuanto a espacio, a uno de estos tres problemas (véase [23]).

En el contexto de lenguajes programación estos problemas tienen gran relevancia. La verificación de tipos, por ejemplo, permite prevenir errores de programación, por lo que muchos lenguajes implementan un mecanismo que les permita verificar que la asignación de un tipo A a un programa t dentro de un contexto de variables locales Γ sea correcto. La asignación de tipos consiste en implementar un algoritmo que permita determinar los tipos de un programa, por lo regular este problema es más difícil que la verificación de tipos porque el algoritmo debe operar correctamente sin la necesidad de especificaciones de tipo del programador, para lograr ésto el algoritmo debe resolver ecuaciones de tipos. El problema de pertenencia desde el punto de vista de un programador puede ser visto como: un tipo vacío, es decir un tipo que no puede ser asignado a ningún término, es una especificación que no puede ser completada por niguna

frase de programa. Resolver el problema de pertenencia signfica la habilidad de excluir tales especificaciones en tiempo de compilación.

En algunos sistemas, como en el presentado en la sección 2.3.2 el problema de la asignación de tipos se reduce al problema de verificación de tipos (una demostración pueder ser consultada en [23]). En otros sistemas estos problemas resultan indecidibles. En el siguiente capítulo se dará otra interpretación a estos problemas en el contexto de la lógica.

2.4. Normalización de la β-reducción

En la sección 2.1.5 se mencionó que en el calculo lambda puro no todo término tenía una forma normal. Pero ya que hemos presentado al cálculo lambda con tipos simples una pregunta interesante es: ¿todo término del cálculo lambda con tipos simples *a la Church* tiene una forma normal?. La respuesta es sí, inclusive todo término del cálculo lambda con tipos simples es fuertemente normalizable, lo que significa que no existe una reducción infinita $M = M_1 \rightarrow_\beta M_2 \rightarrow_\beta \ldots$ para todo término M_1, es decir que no importa como evaluemos una expresión bien tipificada, debemos terminar en algún momento. Desde el punto de vista computacional esta propiedad la podemos entender como: todo programa de un lenguaje basado en el cálculo lambda con tipos simples puede no terminar sólamente si hace uso de un ciclo infinito o una llamada recursiva o un tipo de dato circular.

2.5. Extensión con sumas y productos

En las dos últimas secciones presentamos al cálculo lambda con tipos simples. En ambos sistemas todo término era una función. Sin embargo, como nuestro interés se centra en la aplicación de estos sistemas en los lenguajes de programación, y dado que en éstos son necesarias otras estructuras de datos, presentamos en esta sección una extensión al cálculo lambda con tipos *a la Church*, con el propósito de añadir estructuras de datos simples que nos permitan mayor expresabilidad. En el contexto de la lógica, extender el sistema *a la Church* nos facilitará presentar la relación que guarda el cálculo lambda con tipos y LPI en el siguiente capítulo.

Definición 2.17. *Sea U un conjunto numerable, cuyos miembros serán llamados tipos básicos. El conjunto $\mathbb{T}$ de tipos extendidos es el conjunto definido por la gramática*

$$\mathbb{T} ::= U \mid \mathbb{T} \rightarrow \mathbb{T} \mid \mathbb{T} \times \mathbb{T} \mid \mathbb{T} + \mathbb{T} \mid \mathsf{void}$$

Los constructores $\times$ y $+$ tendrán mayor precedencia que $\rightarrow$ por lo que tipos como $A + B \rightarrow B \times C$ deben entenderse como $(A + B) \rightarrow (B \times C)$. Además preservamos las convenciones del sistema anterior.

Definición 2.18. *El conjunto extendido de términos está definido por la siguiente grámatica:*

$$\Lambda_{\mathbb{T}} ::= \ldots \mid \mathsf{fst}\,\Lambda_{\mathbb{T}} \mid \mathsf{snd}\,\Lambda_{\mathbb{T}} \mid \langle\, \Lambda_{\mathbb{T}}, \Lambda_{\mathbb{T}} \,\rangle \mid \mathsf{inl}^{\mathbb{T}}\Lambda_{\mathbb{T}} \mid \mathsf{inr}^{\mathbb{T}}\Lambda_{\mathbb{T}} \mid \mathsf{abort}_{\mathbb{T}}\Lambda \mid \mathsf{case}(\Lambda_{\mathbb{T}}, V.\Lambda_{\mathbb{T}}, V.\Lambda_{\mathbb{T}}) : \mathbb{T}$$

Aquí solo se presentan los nuevos términos, entendiendo que también se consideran los presentados anteriormente; el concepto de contexto se define de manera análoga.

Definición 2.19. *La relación de tipado $\Gamma \vdash e : A$ donde Γ es un contexto, e es un lambda término y A es un tipo, se define recursivamente mediante las siguientes reglas de inferencia:*

$$\frac{}{\Gamma, x : A \vdash x : A}\ \textit{(Var)}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B}\ \textit{(I-Abs)} \quad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash r : A}{\Gamma \vdash tr : B}\ \textit{(E-Abs)}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash r : B}{\Gamma \vdash \langle\, t, r \,\rangle : A \times B}\ \textit{(I-Prod)} \quad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{snd}\, t : B}\ \textit{(E-Prod)} \quad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{fst}\, t : A}\ \textit{(E-Prod)}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{inl}_{A+B}\, t : B + A}\ \textit{(I-Sum)} \quad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{inr}_{B+A}\, t : A + B}\ \textit{(I-Sum)}$$

$$\frac{\Gamma \vdash t : A + B \quad \Gamma, x : A \vdash r : C \quad \Gamma, y : B \vdash s : C}{\Gamma \vdash \mathsf{case}(t, x.r, y.s) : C}\ \textit{(E-Sum)}$$

$$\frac{\Gamma \vdash x : \mathsf{void}}{\Gamma \vdash \mathsf{abort}_A\ x : A}\ \textit{(void)}$$

Supongamos que $A, B \in \mathbb{T}$, un elemento de $A + B$ es un elemento de A o de B junto con un indicador de a cual pertenece; void es el tipo vacío; $A \times B$ es el tipo producto cuyos elementos son parejas $\langle\, x, y \,\rangle$ donde $y \in B$ y $x \in A$. Con respecto a los nuevos lambda términos tenemos que inl y inr son funciónes de inyección, el término $\mathsf{case}(t, x.r, x_2.s) : C$ puede ser visto como un análisis de casos sobre el término t para determinar un término de tipo C.

Ejemplo 2.16. *Veamos algunos ejemplos*

- $\Gamma = \{x : A \times B\}$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathsf{fst}\, x : A \qquad \dfrac{\Gamma, y : A \vdash \mathsf{snd}\, x : B}{\Gamma \vdash \lambda y : A.\, \mathsf{snd}\, x : A \to B}\,(I - Abs)}{\Gamma \vdash \langle\, \mathsf{fst}\, x, \lambda y : A.\, \mathsf{snd}\, x \,\rangle : A \times (A \to B)}\,(I - Prod)}{\vdash \lambda x : A \times B.\, \langle\, \mathsf{fst}\, x, \lambda y : A.\, \mathsf{snd}\, x \,\rangle : (A \times B) \to (A \times (A \to B))}\,(I - Abs)$$

- $\Gamma = \{f : (A + B) \to C, x : A\}$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash x : A}{\Gamma \vdash \mathsf{inl}_{A+B}\, x : B + A}\,(I - Sum) \qquad \Gamma \vdash f : (A + B) \to C}{\Gamma \vdash f(\mathsf{inl}_{A+B}) : C}\,(E - Sum)}{\vdash \lambda f : (A + B) \to C.\lambda x : A.f(\mathsf{inl}_{A+B}) : ((A + B) \to C) \to (A \to C)}\,(I - Abs)$$

2.5.1. Reducción

Nótese que al agregar nuevos términos al cálculo lambda con tipos simples obtenenmos nuevas reglas de reducción, las cuales definimos a continuación.

Definición 2.20. *La relación de* beta-reducción *sobre los términos extendidos se define como la más pequeña relación compatible* $\to_\beta$ *que extiende la beta-reducción ordinaria de la siguiente manera:*

- $\mathsf{fst}\,\langle\, t, r\,\rangle \to_\beta t$
  $\mathsf{snd}\,\langle\, t, r\,\rangle \to_\beta r$
- $\mathsf{case}\ (\mathsf{inl}\, t, x.r, y.s) \to_\beta r[x := t]$
  $\mathsf{case}\ (\mathsf{inr}\, t, x.r, y.s) \to_\beta s[y := t]$

Observación 2.3. *Se preservan todas las propiedades enunciadas para el cálculo lambda con tipos simples.*

En este capítulo se dió una breve introducción a los sistemas de tipos así como la presentación del cálculo lambda puro, del cual se estudió parte de su poder expresivo y su cercanía con los lenguajes de programación. Lo anterior nos ha servido principalmente para introducir el cálculo lambda con tipos simples en dos estilos *a la Curry* y a *la Church*, además de una extensión con sumas y productos del cálculo lambda *a la Church*. En resumen en este capítulo se han presentado los siguientes sistemas: Cálculo lambda puro (λ), cálculo lambda con tipos simples *a la Curry* ($\lambda^{\to}_{Curry}$), cálculo lambda con tipos simples *a la Church* ($\lambda^{\to}_{Church}$), y *a la Church* con tipos extendidos ($\lambda^{\to,\times,+}_{Church}$). Por lo tanto estamos en condiciones de presentar el tema central de esta tesis el *isomorfismo de Curry-Howard*, que relaciona dos formalismos presentados: el sistema DN para LPI y $\lambda^{\to,\times,+}_{Church}$.

Capítulo 3

El isomorfismo de Curry-Howard

En 1958 Haskell B. Curry, observó que ciertos *combinadores con tipos* podían ser vistos como representaciones de pruebas de ciertas proposiciones (ver [17]). A partir de esa observación Howard, en 1969, describió una correspondencia que asocia proposiciones, demostraciones en deducción natural y normalización de demostraciones con tipos, lambda términos y normalización de la β-reducción, respectivamente. Esta correspondencia se conoce hoy como el *isomorfismo de Curry-Howard*, el cual vincula a las demostraciones de la lógica con la noción de cómputo. Como una consecuencia de este hecho, se han desarrollado varios lenguajes de especificación formal como la familia de lenguajes AUTOMATH desarrollada por N.G. de Bruijn en los años sesenta cuyos descendientes actuales son los sistemas de manejo de demostraciones formales automatizados basados en teorías de tipos como Coq e Isabelle (ver [33] y [32]).

Para algunos, en especial Wadler (véase [25]), esta correspondencia está a la par con la relación entre la teoría de la relatividad de Einstein y la física cuántica de Dirac, por su elegancia e importancia, con la diferencia de que en nuestro caso las matemáticas son mucho más simples.

Antes de pasar a la descripción formal veamos un ejemplo. Considérense las siguientes derivaciones en la lógica proposicional intuicionista del capítulo uno y en el cálculo lambda extendido del capítulo dos, respectivamente.

$\Gamma = \{A \wedge B\}$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,(\wedge E) \qquad \dfrac{\dfrac{\Gamma, A \vdash A \wedge B}{\Gamma, A \vdash B}\,(\wedge E)}{\Gamma \vdash A \to B}\,(\to I)}{\Gamma \vdash A \wedge (A \to B)}\,(\wedge I)}{\vdash (A \wedge B) \to (A \wedge (A \to B))}\,(\to I)$$

$\Gamma = \{x : A \times B\}$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash x : A \times B}{\Gamma \vdash \mathsf{fst}\, x : A}\,(E-Prod) \quad \dfrac{\dfrac{\Gamma, y : A \vdash x : A \times B}{\Gamma, y : A \vdash \mathsf{snd}\, x : B}\,(E-Prod)}{\Gamma \vdash \lambda y : A.\, \mathsf{snd}\, x : A \to B}\,(I-Abs)}{\Gamma \vdash \langle\, \mathsf{fst}\, x, \lambda y : A.\, \mathsf{snd}\, x \,\rangle : A \times (A \to B)}\,(I-Prod)}{\vdash \lambda x : A \times B.\, \langle\, \mathsf{fst}\, x, \lambda y : A.\, \mathsf{snd}\, x \,\rangle : (A \times B) \to (A \times (A \to B))}\,(I-Abs)$$

Tomemos la raíz de la segunda derivación, quedémonos sólo con el tipo, es decir $(A \times B) \to (A \times (A \to B))$ y cambiemos $\times$ por $\wedge$, lo que resulta es $(A \wedge B) \to (A \wedge (A \to B))$, que es una fórmula válida en LPI, de hecho una demostración de élla es la primera derivación. Nótese además que la derivación de esta fórmula es en cierto sentido *igual* a la derivación del término $\lambda x : A \times B.\, \langle\, \mathsf{fst}\, x, \lambda y\ :\ A.\, \mathsf{snd}\, x \,\rangle$. Este término también nos dice como construir una derivación para la fórmula obtenida, puesto que cada constructor de término corresponde a la aplicación de una regla de tipos. De esta manera el término lambda puede ser visto como una codificación de la primera derivación.

Esta relación entre derivación de tipos y demostraciones en deducción natural, es lo que se conoce como el *Isomorfismo de Curry-Howard*. El cual formalizamos a continuación.

3.1. El isomorfismo de Curry-Howard

El isomorfismo o correspondencia de Curry-Howard es un principio fundamental en la teoría de tipos. De manera superficial este isomorfismo establece que hay una correspondencia entre proposiciones y tipos, de tal manera que las demostraciones o derivaciones de una proposición corresponden a programas con un tipo dado. Es decir, dada una proposición A existe un tipo $A^\star$ tal que a cada derivación o demostración de A le corresponde una expresión o término e de tipo $A^\star$. Entre otras cosas esta correspondencia nos dice que las demostraciones tienen un contenido computacional y que los programas son una forma de demostración. Además sugiere que las características de los lenguajes de programación pueden generar conceptos en la lógica, e inversamente. Es importante destacar que esta correspondencia que inició como una modesta observación acerca de tipos y lógicas, se ha desarrollado hasta convertirse en un principio central del diseño de lenguajes de programación y sus implicaciones aún estan siendo exploradas. Si bien el isomorfismo original se da para la lógica minimal, en la actualidad se ha extendido a diversas lógicas, de manera que podría decirse que existen muchas correspondencias de Curry-Howard, de las cuales la presentada aquí es para el cálculo lambda *a la Church* con tipos extendidos.

Definición 3.1. *Sea A una fórmula. Definimos el tipo asociado a A, denotado $A^\star$ como sigue:*

$$\begin{array}{rcl} \bot^\star & = & \mathsf{void} \\ (A \wedge B)^\star & = & A^\star \times B^\star \\ (A \vee B)^\star & = & A^\star + B^\star \\ (A \to B)^\star & = & A^\star \to B^\star \end{array}$$

Se observa que la transformación $\star$ es biyectiva por lo que tiene una inversa denotada $\#$. En particular si A es una fórmula y B es un tipo entonces $(A^\star)^\# = A$ y $(B^\#)^\star = B$.

Dado un contexto lógico $\Gamma = \{A_1, \ldots, A_n\}$ definimos el contexto de tipos $\Gamma^\star$ como $\Gamma^\star = \{x_1 : A_1^\star, \ldots, x_n : A_n^\star\}$. Análogamente dado un contexto de tipos $\Gamma = \{x_1 : B_1, \ldots, x_n : B_n\}$ definimos el contexto lógico $\Gamma^\#$ como $\Gamma^\# = \{B_1^\#, \ldots, B_n^\#\}$.

Ahora estamos listos para enunciar el isomorfismo.

Teorema 3.1 (Isomorfismo de Curry-Howard)**.** *Sean A una fórmula, B un tipo, Γ un contexto lógico y Δ un contexto de tipos.*

1. *Si $\Gamma \vdash A$ entonces existe un término e tal que $\Gamma^\star \vdash e : A^\star$*
2. *Si $\Delta \vdash t : B$ entonces $\Delta^\# \vdash B^\#$*

Demostración. .

1. Inducción sobre la derivación $\Gamma \vdash A$

   - Caso Base
   $$\frac{}{\Gamma, A \vdash A}\ (\text{Hip})$$
   Consideremos $(\Gamma, A)^\star$, usando (Var) tenemos $\Gamma^\star, x_n : A^\star \vdash x_n : A$, por lo tanto tenemos $e = x_n$
   - Si la derivación termina en:
   $$\frac{\Gamma, B \vdash A}{\Gamma \vdash B \to A}\ (\to I)$$
   Por hipótesis de inducción sabemos que existe un término t tal que $\Gamma^\star, x : B^\star \vdash t : A^\star$. Usando $(I\text{-}Abs)$ obtenemos $\Gamma \vdash \lambda x : B^\star.t : B^\star \to A^\star$, de donde $e = \lambda x : B^\star.t$.
   - Si la derivación termina en:
   $$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\to E)$$
   Por hipótesis de inducción sabemos que existen términos t y r tal que $\Gamma^\star \vdash t : A^\star \to B^\star$ y $\Gamma^\star \vdash r : A^\star$. Usando $(E\text{-}Abs)$ obtenemos $\Gamma^\star \vdash tr : B^\star$, y definimos $e = tr$.

- Si la derivación termina en:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,(\vee I)$$

Por hipótesis de inducción tenemos que existe t tal que $\Gamma^\star \vdash t : A^\star$. Usando (*I-Sum*) obtenemos $\Gamma^\star \vdash \mathsf{inl}_{A^\star + B^\star}\, t : B^\star + A^\star$, donde $e = \mathsf{inl}_{A^\star + B^\star}\, t$.

El otro caso para $(\vee E)$ es análogo.

- Si la derivación termina en:

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\,(\vee E)$$

Por hipótesis de inducción sabemos que existen t, r, s tal que $\Gamma \vdash t : (A \vee B)^\star$ es decir $\Gamma \vdash t : A^\star + B^\star$, $\Gamma^\star, x : A^\star \vdash r : C^\star$ y $\Gamma^\star, y : A^\star \vdash s : C^\star$. Usando (*E-Sum*) tenemos $\Gamma^\star \vdash \mathsf{case}(t, x.r, y.s) : C^\star$. Así definimos $e = \mathsf{case}(t, x.r, y.s)$

- Si la derivación término en:

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,(\wedge E)$$

Por hipótesis de inducción sabemos existe t tal que $\Gamma^\star \vdash t : (A \wedge B)^\star$, usando $(E - Prod)$ sabemos que de $\Gamma \vdash \mathsf{fst}\ t : A^\star$, donde $e = \mathsf{fst}\ t$.

De manera análoga para el otro caso de $(\wedge E)$

- Si la derivación término en:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,(\wedge I)$$

Por hipótesis de inducción sabemos que existen t y r tal que $\Gamma^\star \vdash t : A^\star$ y $\Gamma^\star \vdash r : B^\star$, usando $(I - Prod)$ tenemos que $\Gamma^\star \vdash \langle\, t, r \,\rangle : A^\star \times B^\star$, donde $e = \langle\, t, r \,\rangle$.

De manera análoga para el otro caso de $(\wedge I)$.

- Si la derivación termina en:

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\,(\bot E)$$

Por hipótesis de inducción sabemos que existe t tal que $\Gamma^\star \vdash t : \mathsf{void}$, usando (void) tenemos que $\Gamma^\star \vdash \mathsf{abort}_A\ t : A$, donde $e = \mathsf{abort}_A\ t : A$

2. Inducción sobre la derivación $\Delta \vdash t : B$

- Case base

$$\frac{}{\Gamma, x_n : A \vdash x_n : A}\,(\text{Var})$$

Por definición $\{\Gamma, x_n : A\}^\#$ es igual a: $\Gamma^\#, A^\#$, y por lo tanto tenemos $\Gamma^\#, A^\# \vdash A^\#$ usando (Hip).

- Si la derivación termina en:

$$\frac{\Delta, x : A \vdash t : B}{\Delta \vdash \lambda x : A.t : A \to B}$$

  Por hipótesis de inducción sabemos: $(\Delta, x : A)^{\#} \vdash B^{\#}$, es decir $\Delta^{\#}, A^{\#} \vdash B^{\#}$, por lo que, usando la regla $(\to I)$ tenemos $\Delta^{\#} \vdash A^{\#} \to B^{\#}$, es decir, $\Delta^{\#} \vdash (A \to B)^{\#}$.

- Si la derivación termina en:

$$\frac{\Delta \vdash t : A \to B \quad \Delta \vdash r : A}{\Delta \vdash tr : B}$$

  Por hipótesis de inducción sabemos: $\Delta^{\#} \vdash (A \to B)^{\#}$ es decir $\Delta^{\#} \vdash A^{\#} \to B^{\#}$; además sabemos $\Delta^{\#} \vdash A^{\#}$. Usando $(\to E)$ tenemos $\Delta^{\#} \vdash B^{\#}$.

- Si la derivación termina en:

$$\frac{\Delta \vdash t : A \quad \Delta \vdash r : B}{\Delta \vdash \langle t, r \rangle : A \times B}$$

  Por hipótesis de inducción sabemos $\Delta^{\#} \vdash A^{\#}$ y $\Delta^{\#} \vdash B^{\#}$. Si usamos $(\wedge I)$ tenemos $\Delta^{\#} \vdash A^{\#} \wedge B^{\#}$ que por definición es: $\Delta^{\#} \vdash (A \times B)^{\#}$

Los demás casos son análogos.

□

3.2. Contenido computacional y codificación de las derivaciones

Dada una derivación lógica $\Gamma \vdash A$ el teorema 3.1 garantiza la existencia de una expresión e y de una derivación de tipos $\Gamma^{\star} \vdash e : A^{\star}$ y viceversa. Más aún la demostración de este teorema nos deja ver que la estructura de ambas derivaciones es idéntica. Este hecho justifica formalmente la afirmación de que las demostraciones lógicas tienen un contenido computacional, a saber la demostración lógica $\Gamma \vdash A$ tiene contenido computacional e, puesto que ambas derivaciones son esencialmente la misma. Esto implica que la derivación $\Gamma^{\star} \vdash e : A^{\star}$ reciba dos lecturas distintas, si omitimos el uso de la operación $\star$, a saber:

- Lógicamente: $\Gamma \vdash e : A$ significa que la fórmula A es derivable a partir de las hipótesis Γ y e es un código de demostración para dicha derivación.
- Computacionalmente: $\Gamma \vdash e : A$ significa que e es un programa con tipo A y variables locales en Γ.

Más aún, visto lógicamente el código de demostración e permite definir de manera formal los principios constructivos dados por la intepretación BHK descrita en la definición 1.4.

Veamos con detalle estas afirmaciones para cada operador lógico.

3.2.1. La implicación

Introducción de la implicación

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.e : A \to B}$$

El código de demostración es $\lambda x : A.e$ que representa a una definición de la función $x \mapsto e$, donde usualmente x figura en e. Bajo la interpretación BHK, este código o programa es precisamente el método funcional que transforma a cada derivación x de A en una derivación e de B.

Eliminación de la implicación

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash t : A}{\Gamma \vdash ft : B}$$

El código de demostración f es un método funcional que transforma cada derivación de A en una derivación de B; t es el código de una demostración particular de A, por lo tanto bajo la interpretación BHK, el código de demostración ft es la aplicación de la función f a la demostración t de A que, por definición de f, transformará a t en una demostración de B.

En programación funcional f es un programa que recibe como parámetro el programa t, de tipo A y que regresa como salida una expresión de tipo B que representa la evaluación de f en t.

3.2.2. La conjunción

Introducción de la conjunción

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash r : B}{\Gamma \vdash \langle t, r \rangle : A \times B}$$

El código de demostración $\langle t, r \rangle$ bajo la interpretación BHK es una construcción de t y r, que son códigos de demostración de A y B respectivamente. En programación funcional esta construcción representa a un procedimiento que permite construir un programa a partir de dos programas previos donde el nuevo programa será del tipo $A \times B$.

Eliminación de la conjunción

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{fst}\, t : A}$$

El código de demostración de t bajo la interpretación BHK es una construcción de una demostración de A y B , por lo que el código de demostración $\mathsf{fst}\, t$ es un procedimiento que nos permite obtener el código de demostración de A, a partir de de t. De manera análoga para $\mathsf{snd}\, t$. Esta regla nos permitiría obtener de algún programa de tipo $A \times B$ su primer componente, que sería un programa de tipo A. La regla para snd es semejante a ésta y nos permite obtener la segunda componente, es decir un programa de tipo$B$.

3.2.3. La disyunción

Introducción de la disyunción

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{inl}_{A+B}\, t : B + A}$$

La expresión $\mathsf{inl}\, t$ corresponde, bajo la interpretación BHK, al par $(1, t)$, donde t es un código de demostración para A. Análogamente $\mathsf{inr}\, t$ corresponde al par $(2, t)$, donde t es un código de demostración de B. En lenguajes de programación esta regla corresponde a la inyección de valores de un tipo en un tipo más general, esto se conoce en programación funcional como un tipo opción o variante.

Eliminación de la disyunción

$$\frac{\Gamma \vdash t : A + B \quad \Gamma, x : A \vdash r : C \quad \Gamma, x_2 : B \vdash s : C}{\Gamma \vdash \mathsf{case}(t, x.r, x_2.s) : C}$$

t es un código de demostración de $A \vee B$; $\lambda x : A.r$, y $\lambda x_2 : B.s$ son métodos funcionales que transforman a cada derivación de A o B en una de C respectivamente, por lo que el código de demostración $\mathsf{case}(t, x.r, x_2.s)$ es un programa que evalua si t es un código de demostración de A o de B, y lo transforma en una derivación de C usando los métodos correspondientes.

En lenguajes de programación esta regla la podemos ver como una evaluación por casos, el término o condición a evaluar es t, de tipo $A + B$, en caso de que el valor obtenido pertenezca a A se asigna a t el valor de r, en caso contrario se le asigna el valor de s, como ambos son del tipo C siempre devuelve algo de este tipo.

3.2.4. El falso

Elminación de void

$$\frac{\Gamma \vdash t : \mathsf{void}}{\Gamma \vdash \mathsf{abort}_A\, t : A}$$

Esta regla computacionalmente puede ser vista como el lanzamiento de una excepción sin manejo, es decir en el caso de suceder algun evento inesperado, lo cual se indica al derivar void, se interrumpe el flujo normal del programa devolviendo como resultado una expresión $\mathsf{abort}_{\mathsf{t}}$ que podría ser de cualquier tipo A.

Ejemplo 3.1. *Veamos finalmente un par de ejemplos de codificación de demostraciones.*

- *El lambda término $\lambda x : A.x$ de tipo $A \to A$, codifica una demostración de $A \to A$, de la siguiente manera: Del cuerpo de la lambda abstracción, sabemos que debemos usar nuestra hipótesis codificada por x, la lambda abstracción nos dice que después debemos usar la regla de introducción de la implicación, descargando nuestra única hipótesis y obteniendo la demostración buscada, es decir:*

$$\frac{A \vdash A}{\vdash A \to A}\,(\to I)$$

- *Una demostración de $A \to (A \to B) \to B$ se encuentra codificada por el lambda término: $\lambda y : A.\lambda x : A \to B.xy$. Codificando nuestras hipótesis de la siguiente manera: $A \to B$ con x, y y para A. Una demostración codificada de B, dadas nuestra hipótesis, se encuentra en el cuerpo de la lambda abastracción más a la derecha, la cual es la aplicación xy, por lo que debemos usar la regla de eliminación de la implicación, con nuestras dos hipótesis, para obtener B. Lo que resta es hacer uso de la regla de introducción de la implicación dos veces, descargando primero nuestra hipótesis codificada con x, y al final la codificada con y. La derivación resultante es:*

$$\dfrac{\dfrac{\dfrac{A \to B, A \vdash A \to B \qquad A \to B, A \vdash A}{A \to B, A \vdash B}\,(\to E)}{A \vdash (A \to B) \to B}\,(\to I)}{\vdash A \to (A \to B) \to B}\,(\to I)$$

Después de la discusión y ejemplos anteriores es claro que los lambda términos que reciben un tipo de acuerdo a las reglas dadas proporcionan una notación lineal para las derivaciones lógicas correspondientes.

Otras observaciones interesantes del isomorfismo de *Curry-Howard* son las siguientes: Las codificaciones, pueden ser vistas como las construcciones de la interpretación BHK de LPI; a cada construcción le corresponde una sola proposición, pero como una proposición puede ser demostrada de distintas formas, puede que haya mas de una construcción por cada proposición; los tipos corresponden a las fórmulas de la LPI, y los constructores a los conectivos.

Considerese los siguientes problemas en el cálculo lambda:

- Dado $A \in \mathbb{T}$ decidir si existen Γ y t tal que $\Gamma \vdash t : A$ *(inhabitation)*
- Dado t, decidir si existen Γ y A tal que $\Gamma \vdash t : A$ (Asignación de tipos)

Son equivalentes,respectivamente, a los siguientes problemas en la lógica:

- Verificar si una fórmula es demostrable.
- Dada una codificación t, decidir si corresponde a una prueba válida. En otras palabras decidir si hay una fórmula A tal que t codifica una prueba de A.

3.3. Normalización y $\beta-$reducción

En el capítulo uno se habló brevemente sobre la normalización de demostraciones; se dijo que la eliminación de redundancias en éllas ayuda a su estudio. Una ves presentado el *isomorfismo de Curry-Howard* la pregunta obligada es ¿a qué corresponde el concepto de normalización de demostraciones bajo el este isomorfismo? para responder esta pregunta veamos unos ejemplos.

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \vdash A & \Gamma \vdash B\end{array}}{\Gamma \vdash A \wedge B}\,(\wedge I)}{\Gamma \vdash A}\,(\wedge E)$$

Esta es una derivación de A redundante, ya que después de usar $(\wedge I)$ usamos $(\wedge E)$, por lo que podemos encontrar una demostración más sencilla, por ejemplo:

$$\begin{array}{c}\vdots \\ \Gamma \vdash A\end{array}$$

Por lo tanto, podriamos decir que la primera derivación se reduce o simplifica a la segunda (para decir que D_1 se simplifica a D_2, donde D_1 y D_2 son derivaciones escribimos $D_1 \to D_2$) es decir:

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \vdash A & \Gamma \vdash B\end{array}}{\Gamma \vdash A \wedge B}\,(\wedge I)}{\Gamma \vdash A}\,(\wedge E) \qquad \to \qquad \begin{array}{c}\vdots \\ \Gamma \vdash A\end{array}$$

Usando la codificación de estas demostraciones mediante lambda términos obtendriamos:

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \vdash t : A & \Gamma \vdash r : B\end{array}}{\Gamma \vdash \langle t, r \rangle : A \times B}\,(I - Prod)}{\Gamma \vdash \mathsf{fst}\,\langle t, r \rangle : A}\,(E - Prod) \qquad \to \qquad \begin{array}{c}\vdots \\ \Gamma \vdash t : A\end{array}$$

Es decir la regla de reducción: $\mathsf{fst}\,\langle t, r\,\rangle \to t$.

Veamos otro ejemplo.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, A \vdash B\end{array}}{\Gamma \vdash A \to B}\,(\to I) \qquad \begin{array}{c}\vdots\\ \Gamma \vdash A\end{array}}{\Gamma \vdash B}\,(\to E)$$

Esta derivación nos dice que hay una prueba de B con hipótesis Γ y A, pero adicionalmente hay una prueba de A con hipótesis Γ entonces hay una prueba de B con hipótesis Γ, lo cual nos da la regla derivada $(Sust)$.

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma, A \vdash B\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma \vdash A\end{array}}{\Gamma \vdash B}\,(Sust)$$

De donde notamos que la hipótesis A en $\Gamma, A \vdash B$ es redundante. Así la demostración $\Gamma \vdash B$ se construye a partir de la demostración $\Gamma, A \vdash B$ sustituyendo todas las presencias de A por la demostración $\Gamma \vdash A$. Esto se hace explícito cuando introducimos anotaciones:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, x : A \vdash t : B\end{array}}{\Gamma \vdash \lambda x : A.t : A \to B}\,(I - Abs) \qquad \begin{array}{c}\vdots\\ \Gamma \vdash r : A\end{array}}{\Gamma \vdash (\lambda x : A.t)r : B}\,(E - Abs)$$

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma, x : A \vdash t : B\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma \vdash r : A\end{array}}{\Gamma \vdash t[x := r] : B}\,(Der)$$

Como cada regla de introducción introduce un constructor (una lambda abstracción, un par o una inyección) y cada regla de eliminación introduce un *destructor* (aplicación, proyección, etc) entonces un *redex* corresponde al uso de una rxegla de introducción seguida de la correspondiente regla de eliminación. Por lo tanto las reducciones en los términos corresponden a la normalización de demostraciones.

Observación 3.1. *Las demostraciones que no están en forma normal son demostraciones sin sentido desde el punto de vista puramente lógico, pero que al tomar en cuenta al isomorfismo de Curry Howard del lado computacional corresponden al mecanismo de evaluación de un rudimentario lenguaje de programación funcional.*

3.4. Curry-Howard y programación

Desde un punto de vista más pragmático, y para resaltar al cálculo lambda como fundamento de los lenguajes de programación funcionales veremos como el isomorfismo de *Curry-Howard* nos permite visualizar a los sistemas de deducción natural como una especie de lenguaje de programación funcional, donde las fórmulas de la lógica corresponden a los tipos del lenguaje, las demostraciones o derivaciones corresponden a programas del mismo; y donde verificar si una demostración es correcta, en un contexto dado, es equivalente a verificar que un programa tiene un tipo legal en el contexto. Por lo tanto si en la lógica $\Gamma \vdash M : A$ significa que A es derivable a partir de Γ, y M es una codificación de tal derivación; en computación significa que M es un programa de tipo A con variables locales en Γ.

Veamos algunos ejemplos más. No se escriben las derivaciones completas, ya que los programas indican los pasos que se deben realizar para obtener el tipo asignado.

- $\vdash \lambda x : A.\, \mathsf{inl}_{A+B}\, x : A \to A \vee B$
- $\vdash \lambda x : A \wedge B.\, \mathsf{snd}\, x : A \wedge B \to B$
- $P : A \to B,\, Q : B \to C \vdash \lambda x : A.Q(Px) : A \to C$
- $\vdash \lambda x : B.\lambda y : A.x : B \to (A \to B)$
- $\vdash M : (A \wedge B \to C) \to (A \to B \to C)$
- $\lambda f : A \wedge B \to C.\lambda x : A.\lambda y : B.f\, \langle\, x, y \,\rangle : (A \wedge B \to C) \to (A \to B \to C)$
- $f : A \to B \to C \vdash \lambda x : A \wedge B.(f(\mathsf{fst}\, x))(\mathsf{snd}\, x) : A \wedge B \to C$
- $f : A \vee B \to C \vdash \lambda x : A.f(\mathsf{inl}_{A+B}\, x) : A \to C$
- $x : A \vee B, f : A \to C, g : B \to C \vdash (\mathsf{case}\ \ x, y.fy, z.gz) : C$
- $f : A \to B, g : C \to D, x : A \vee C \vdash (\mathsf{case}\ \ x, y.\, \mathsf{inl}_{B+D}\, fy, z.\, \mathsf{inr}_{B+D}\, gz) :$ $B \vee D$
- $x : (A \wedge B) \vee (A \wedge C) \vdash (\mathsf{case} x, y.\, \langle\, \mathsf{fst}\, y, \mathsf{inl}_{B+C}(\mathsf{snd}\, y) \,\rangle, z.\, \langle\, \mathsf{fst}\, z, \mathsf{inr}_{B+C}(\mathsf{snd}\, z)) \,\rangle :$ $A \wedge (B \vee C)$
- $x : A \wedge (B \vee C) \vdash (\mathsf{case}\, \mathsf{snd}\, x, y.\, \mathsf{inl}_{(A\times B)+(A\times C)}\, \langle\, \mathsf{fst}\, x, y \,\rangle; z.\, \mathsf{inr}_{(A\times B)+(A\times C)}\, \langle\, \mathsf{fst}\, x, z \,\rangle) :$ $(A \wedge B) \vee (A \wedge C)$

Para finalizar mostramos una clase especial de derivaciones redundantes cuyo contenido computacional es de suma importancia.

3.4.1. Numerales de Church

Consideremos las siguientes derivaciones

$$\dfrac{\dfrac{\dfrac{p \to p, p \vdash p}{p \to p \vdash p \to p}\,(\to I)}{\vdash (p \to p) \to p \to p}\,(\to I)}$$

$$\dfrac{\dfrac{\dfrac{p \to p, p \vdash p \qquad p \to p, p \vdash p \to p}{p \to p, p \vdash p}\,(\to E)}{p \to p \vdash p \to p}\,(\to I)}{\vdash (p \to p) \to p \to p}\,(\to I)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{p \to p, p \vdash p \qquad p \to p, p \vdash p \to p}{p \to p, p \vdash p}\,(\to E) \qquad p \to p, p \vdash p \to p}{p \to p, p \vdash p}\,(\to E)}{p \to p \vdash p \to p}\,(\to I)}{\vdash (p \to p) \to p \to p}\,(\to I)$$

Estas derivaciones parecen no tener mucho sentido, ya que son redundantes. Sin embargo bajo el isomorfismo de *Curry-Howard* codificando las hipótesis de la siguiente manera: $f : p \to p$ y $x : p$. Obtendríamos lo siguiente:

- $\lambda f : p \to p.\lambda x : p.x : (p \to p) \to p \to p$
- $\lambda f : p \to p.\lambda x : p.fx : (p \to p) \to p \to p$
- $\lambda f : p \to p.\lambda x : p.ffx : (p \to p) \to p \to p$
- $\lambda f : p \to p.\lambda x : p.f^n x : (p \to p) \to p \to p$

Que son, recordando la sección 2.2, los numerales de Church. Por lo tanto existen derivaciones en la lógica que a pesar de ser redundantes y de no tener un uso importante, gracias al *Isomorfismo de Curry Howard* toman gran importancia.

3.5. Resumen

En la siguiente tabla se resume lo más importante de este capítulo

λ	LPI
Variable de Tipo	Variable proposicional
Tipo	fórmula
Constructor de tipo	Conectivo
Asignación de tipo	Demostrabilidad
Redex	Demostración redundante
Normalización de β-reducción	Normalización de pruebas

El tema central de esta tesis ha sido discutido, desde un punto de vista teórico principalmente. Sin embargo es de nuestro interés mostrar desde del punto de vista práctico la importancia del *isomorfismo de Curry-Howard* en los lenguajes de programación funcionales. En el siguiente capítulo se presenta la descripción e implementación de un pequeño lenguaje basado en gran medida en el isomorfismo presentado.

Capítulo 4

MinHaskell

En este capítulo presentamos de manera formal un pequeño lenguaje de programación al cual llamamos MinHaskell, escrito en Haskell, un lenguaje de programación funcional y de evaluación perezosa. MinHaskell esta basado en Minml de Frank Pfenning (véase [10]).

4.1. Programación funcional

El paradigma de programación funcional ha crecido en popularidad durante las últimas décadas, desde sus inicios con los primeros dialectos de LISP al día de hoy con la disponibilidad de diversos lenguajes de propósito múltiple, como Standard ML y Haskell.
Los lenguajes funcionales son usados cada vez con más frecuencia como componentes de sistemas más amplios debido a su facilidad de interacción, la cual usualmente se realiza sin sacrificar su elegancia semántica. Además proporcionan un marco donde las ideas cruciales de la programación moderna se presentan de la manera más clara posible. Esto se refleja mediante el amplio uso en la enseñanza en ciencias de la computación y en su influencia en el diseño de otros lenguajes. Un caso relevante es el diseño de G-Java, cuyos mecanismos genéricos se modelan directamente con el polimorfismo de Haskell.

Aunque existen diferencias entre los diversos lenguajes de programación funcional, también hay un amplio consenso acerca de las características principales de los mismos, las cuales mencionamos a continuación:

- Funciones de primera clase: Las funciones pueden ser pasadas como argumentos, y devueltas como resultados de otras funciones; un ejemplo es la función `map`, la cual toma una función, digamos f, como argumento, y devuelve la función que toma una lista, xs, como argumento y devuelve la lista resultante de aplicar f a cada elemento en xs.
- Sistemas fuertemente tipados: El lenguaje contiene distinciones entre di-

ferentes valores, clasificando valores similares en tipos. La tipificación de valores restringe la aplicación de operadores y constructores de datos, por lo que errores en los cuales, por ejemplo, dos valores booleanos son sumados, no serán permitidos.

- Tipos polimórficos: Una objeción a la tipificación fuerte es que no se puede reutilizar el código como en los lenguajes sin tipos, ya que en ellos el código es independiente del contenido, por ejemplo, el código de la función identidad es el mismo para cualquier tipo. No perder la tipificación fuerte y esta generalidad del código de los lenguajes no tipificados se puede lograr usando un sistema de tipos adecuado como el de Milner, y es lo que se conoce como tipos polimórficos.

- Tipos algebraicos: El mecanismo de tipos algebraicos generaliza los tipos invariantes, registros, y ciertos tipos de definiciones de apuntadores de tipo, además permite que defincciones de tipos, como listas, sean parametrizadas sobre tipos.

- Modularidad: Se provee de sistemas de módulos de diverso grado de complejidad, lo que significa que sistemas grandes pueden ser desarrollados más fácilmente.

Un área en la cual hay diferencias es en el mecanismo de evaluación. SML incorpora evaluación estricta, en el cual los argumentos de las funciones son evaluados antes de evaluar el cuerpo de la función, y los componentes de los tipos de datos son totalmente evaluados sobre la formación del objeto. Por otro lado, Miranda y Haskell incorporan evaluación perezosa, en donde los argumentos de las funciones y los componentes de los tipos de datos son solo evaluados cuando es necesario. Esto crea una distinción en el estilo de programación, los basados en estructuras de datos infinitas o en los que éstas sólo son parcialmente definidas. Hay sistemas que incorporan ambos estilos como Hope+.

Este no es el lugar para dar una introducción completa a la programación funcional, el lector interesado puede consultar [28] y [29]

4.2. Sintaxis

La descripción de la sintaxis de un lenguaje de programación se sirve de dos clases de objetos, las cadenas y los árboles de sintaxis abstracta. Las cadenas proporcionan una representación lineal conveniente para la interacción humana, esto se conoce como *sintaxis concreta*, presentada usualmente mediante gramáticas libres de contexto. Por otra parte los árboles de sintaxis abstracta modelan la estructura jerárquica de la sintaxis y son mucho mas adecuados que las cadenas para el análisis y la mecanización.

4.2.1. Sintaxis concreta

La sintaxis concreta de un lenguaje es un medio para representar las expresiones o cadenas que pueden escribirse en una página de código mediante el uso del teclado. Esta sintaxis usualmente se diseña para mejorar la lectura y eliminar la ambigüedad del código. El método estándar para definir la sintaxis concreta es mediante una gramática libre de contexto donde los símbolos terminales representan los lexemas o *tokens* del lenguaje, las clases sintácticas corresponden a los símbolos no-terminales y las reglas de producción determinan que cadenas de tokens corresponden a que clase sintáctica. A continuación damos la sintaxis concreta de minHaskell:

$$
\begin{array}{lcl}
TipoBase & ::= & Int \mid Bool \mid void \mid unit \mid (Tipo) \\
Tipo & ::= & TipoBase \mid TipoBase \rightarrow Tipo \\
 & & \mid TipoBase \times Tipo \mid TipoBase + Tipo \\
Var & ::= & < identificador > \\
OpAd & ::= & + \mid - \\
OpMu & ::= & * \\
OpRel & ::= & == \mid <
\end{array}
$$

Las gramáticas anteriores nos permiten definir los tipos que nuestros programas en minHaskell pueden tener, los cuales son: Booleanos, enteros, función, tipo producto y tipo suma o invariante. Además se definen categorías auxiliares que nos permitirán hacer uso de operaciones primitivas como la suma resta, multiplicación, y comparación.

$$
\begin{array}{lcl}
FactorA & ::= & (Exp) \mid n \mid Var \mid True \mid False \\
 & & \mid \mathsf{if}\ Exp\ \mathsf{then}\ Exp\ \mathsf{else}\ Exp \\
 & & \mid \mathsf{let}\ Var : Tipo = Exp\ \mathsf{in}\ Exp\ \mathsf{end} \\
 & & \mid \mathsf{fun}\ (Var : Tipo) = Exp \\
 & & \mid \mathsf{fun}\ Var\ (Var : Tipo) : Tipo = Exp \\
 & & \mid \mathsf{case}\ Exp\ \mathsf{of}\ \mathsf{inl}\ \mathsf{Var}\ =>\ Exp \mid \mathsf{inr}\ \mathsf{Var}\ =>\ Exp \\
 & & \mid \mathsf{fst}\ Tipo\ Exp \\
 & & \mid \mathsf{snd}\ Tipo\ Exp \\
 & & \mid \mathsf{inr}\ Tipo\ Exp \\
 & & \mid \mathsf{inl}\ Tipo\ Exp \\
 & & \mid \mathsf{abort}\ Tipo\ Exp \\
 & & \mid \mathsf{triv}
\end{array}
$$

$$
\begin{array}{lcl}
Factor & ::= & FactorA \mid Factor\ Exp \\
Term & ::= & Factor \mid FactorOpMul\ Term \\
Exp' & ::= & Term \mid Term\ OpAd :\ Exp \\
Exp & ::= & Exp' \mid Exp'\ OpRel\ Exp
\end{array}
$$

Exp es la gramática que define a las expresiones o programas de minHaskell, es complicada pero no ambigua lo cual facilitará la implementación de los analizadores léxico y sintáctico.

Ejemplo 4.1. *Algunos ejemplos de programas en MinHaskell.*

- `fun factorial (x:Int):Int = if  x == 1 then 1 else x * factorial (x-1)`

  Este programa define a la función factorial de un número entero positivo. Nótese que se dan de manera explícita los tipos de los argumentos.

- `fun suma (x:Int):Int = x + 2`

  La función suma añade dos unidades a un numero x; esta función es un ejemplo de una función con nombre pero no recursiva.

- `(fun (x:Int) = (x + 4)* 34) 4`

  Esta función es la aplicación de la función anónima que suma 34 a su argumento al cuatro

- `let x:Int = 23 in (fun (y:Int) = x + y) 44 end`

  Este último ejemplo es la declaración de un let típico. Nótese que el tipo de los argumentos debe ser especificado, como en el caso de las funciones recursivas.

4.2.2. Sintaxis abstracta

Cuando se analiza un lenguaje desde el punto de vista matemático se está interesando en la estructura del lenguaje no en su representación, por lo que es necesario tener una sintaxis adecuada para dicha estructura; la sintaxis concreta no es lo que buscamos, ya que suele tener problemas por la complejidad de sus gramáticas y por ser una representación lineal, en su lugar se usa una sintaxis abstracta que expresa la estructura jerárquica de las componentes de un lenguaje.

La sintaxis abstracta proporciona una representación con la cual podemos estudiar el lenguaje mediante un árbol obtenido después de la fase de anáĺisis sintáctico llamado árbol de sintaxis abstracta (asa) cuyos nodos pueden estar etiquetados por un operador. Un operador tiene un índice asignado que indica el número de argumentos que recibe, los cuales, corresponden al número de hijos de cualquier nodo etiquetado con él.

La sintaxis concreta y la sintaxis abstracta se relacionan mediante el proceso de traducción de la sintaxis concreta hacia la abstracta, que se conoce como análisis sintáctico. Un analizador sintáctico deber verificar si un programa es correcto con respecto a la sintaxis concreta y representarlo con un árbol de sintaxis abstracta.

Como se dijo las grámaticas que describen la sintaxis concreta son complicadas, por lo que en MinHaskell usaremos la siguiente grámatica para describir la sintaxis abstracta:

```
Exp ::= VAR < ident >
      | OP Op [Exp]
      | LAM VAR < ident > Tipo Exp
      | IF Exp Exp Exp
      | REC VAR < ident > Tipo Exp
      | CASE Exp VAR < ident > Exp Var Exp
      | AP Exp Exp
      | PAIR Exp Exp
      | FST Exp
      | SND Exp
      | INR Tipo Exp
      | INL Tipo Exp
      | ABORT Tipo Exp
      | TRIV
```

< ident > es un identificador.

Veamos los programas del ejemplo 4.1 escritos en sitaxis abstracta.

Ejemplo 4.2. *Sintaxis abstracta.*

- ```
  REC VAR "factorial" (TINT :>: TINT)
  (LAM VAR "x" TINT (IF (OP Igual [VAR "x",INT 1]) (INT 1)
  (OP Por [VAR "x",AP (VAR "f") (OP Menos [VAR "x",INT 1])])))
  ```
- ```
  REC VAR "suma" (TINT :>: TINT) (LAM VAR "x" TINT (OP Mas [VAR "x",INT 2]))
  ```
- ```
  AP (LAM VAR "x" TINT (OP Por [OP Mas [VAR "x",INT 4],INT 34])) (INT 4)
  ```
- ```
  AP (LAM VAR "x" TINT (AP (LAM VAR "y" TINT (OP Mas [VAR "x",VAR "y"]))
  (INT 44))) (INT 23)
  ```

Los ejemplos 4.1 y 4.2 nos permiten contrastar ambas sintaxis. Escribir programas en sintaxis abstracta no sería nada recomendable, al igual que estudiar los programas en sintaxis concreta.

4.3. Semántica

La semántica de un lenguaje de programación da el significado a diversas características e instrucciones del lenguaje, por ejemplo, el comportamiento en tiempo de ejecución. Es importante dar de manera formal la semántica ya que esto ayuda a evitar errores en el diseño de un lenguaje, además es útil en la optimización del análisis de los programas. La semántica normalmente se presenta en dos niveles:

- Semántica estática: determina cuando un programa está bien definido mediante criterios sintácticos. Por ejemplo verificando si hay presencias libres de variables o si son correctos los tipos asignados a los programas. Esto se lleva a cabo en tiempo de compilación de ahí el adjetivo estática.
- Semántica dinámica: determina el valor o evaluación de un programa. Hay tres estilos para definir la semántica dinámica: denotacional, axiomático y operacional. Nosotros usaremos este último.

4.3.1. Semántica dinámica

En la semántica operacional el significado de una instrucción de programa se define en términos de su comportamiento durante la ejecución. El comportamiento del programa se modela definiendo una máquina abstracta en el sentido de que usa las expresiones del lenguaje como código de máquina. Para lenguajes simples, como MinHaskell, un estado es simplemente una expresión y el comportamiento de la máquina se define mediante una relación o sistema de transición que devuelve el siguiente estado, el cual se obtiene al desarrollar una simplificación, evaluación, o bien declarando que la máquina se ha detenido. El significado de una expresión e es entonces el estado final alcanzado por la máquina si inició su funcionamiento tomando a e como estado inicial. Eso se conoce como semántica estructural o de paso pequeño. En contraste existen las llamadas semánticas naturales o de paso grande que evalúan el término devolviendo el resultado final en un solo paso. En nuestro caso procedemos a definir una semántica operacional de paso pequeño para minHaskell.

Acontinuación definimos una semántica estructural para minHaskell, donde los estados son expresiones cerradas, sin variables libres, todos los cuales son estados iniciales. Los estados finales son los valores definidos por las siguientes reglas, donde e val significa que la expresión e es un valor:

$$\frac{}{\text{(INT n) val}}$$

$$\frac{}{\text{(BOOL b) val}}$$

$$\frac{}{\text{(TRIV) val}}$$

$$\frac{}{\text{(LAM x A e) val}}$$

Estas reglas nos dicen que los números enteros, las constantes booleanas, el término triv y las lambda abstracciones son valores. Podría no parecer natural que una lambda abstracción sea un valor, pero basta recordar que nuestro lenguaje es funcional y que por lo tanto es natural considerar a una función como un objeto básico o valor.

La relación de transición, $e \mapsto e'$, se define inductivamente como sigue:

- Operaciones primitivas

$$\frac{e_i \mapsto e_i'}{\mathsf{OP\ Op}[v_1, .., v_{i-1}, e_i, .., e_n] \mapsto \mathsf{OP\ Op}(v1, .., v_{i-1}, e_i', .., e_n)}$$

$$\frac{v_1\ \mathsf{val}\ \ v_2\ \mathsf{val} \ldots\ v_n\ \mathsf{val}}{\mathsf{OP\ Op}[v_1, v_2, v_3, .., v_n] \mapsto v\ \mathsf{val}}$$

La evaluación de las operaciones primitivas (suma, resta, multiplicación etc) es estricta ya que éstas se evaluan de izquierda a derecha.

- Condicionales

$$\frac{e_1 \mapsto e_1'}{\mathsf{IF}\ \ e_1\ \ e_2\ \ e_3 \mapsto \mathsf{IF}\ \ e_1'\ \ e_2\ \ e_3}$$

$$\frac{}{\mathsf{IF}\ \ (\mathsf{BOOL\ True})\ \ e_2\ \ e_3 \mapsto e_2} \qquad \frac{}{\mathsf{IF}\ \ (\mathsf{BOOL\ False})\ \ e_2\ \ e_3 \mapsto e_3}$$

Estas reglas indican la forma en que se evalúa una expresión IF. Se debe evaluar primero la expresión e_1, que es la condición y por lo tanto se debe evaluar a un valor booleano; una vez que se conoce este resultado se tienen dos casos: e_1 se evalua a BOOL True, o bien a BOOL False. En el primer caso la expresión IF se evaluará a e_2; en el segundo a e_3.

- Aplicaciones

$$\frac{e_1 \mapsto e_1'}{\mathsf{AP}\ \ e_1\ \ e_2 \mapsto \mathsf{AP}\ \ e_1'\ \ e_2} \qquad \frac{v\ \ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathsf{AP}\ \ v\ \ e_2 \mapsto \mathsf{AP}\ \ v\ \ e_2'}$$

$$\frac{v\ \mathsf{val}}{\mathsf{AP}\ \ (\mathsf{LAM}\ \ x\ \ A\ \ e)\ \ v \mapsto e[x := v]}$$

Las reglas de evaluación de las expresiones AP, indican que primero hay que evaluar a la expresión e_1, para luego evaluar la expresión e_2. La tercer regla nos indica que la evaluación de e_1 debe ser una función anónima en donde la variable ligada será substituida por v en e, que es el valor de e_2.

- Pares

$$\frac{e_1\ \mathsf{val}\ e_2\ \mathsf{val}}{\mathsf{PAIR}\ e_1, e_2}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{PAIR}\ e_1\ \ e_2 \mapsto \mathsf{PAIR}\ e_1'\ \ e_2} \qquad \frac{v\ \mathsf{val} \quad e_2 \mapsto e_2'}{\mathsf{PAIR}\ ve_2 \mapsto \mathsf{PAIR}\ v\ \ e_2'}$$

Las reglas indican que se deben determinar los valores de cada una de las componentes de un par de izquierda a derecha.

$$\frac{e \mapsto e'}{\mathsf{FST}\ e \mapsto \mathsf{FST}\ e'} \qquad \frac{v_1\ \mathsf{val}\ \ v_2\ \mathsf{val}}{\mathsf{FST\ PAIR}\ v_1\ \ v_2 \mapsto v_1}$$

Estas reglas nos indican que para obtener el valor de la primera proyección de una par se debe determinar primero el valor de cada componente del par en cuestión, una ves que esto sucede se procede a devolver v_1.

- Sumas

$$\frac{e \mapsto e'}{\mathsf{ABORT}\ A\ e \mapsto \mathsf{ABORT}\ A\ e}$$

$$\frac{e\ \mathsf{val}}{(\mathsf{INL}\ A\ e)\ \mathsf{val}} \qquad \frac{e\ \mathsf{val}}{(\mathsf{INL}\ A\ e)\ \mathsf{val}}$$

$$\frac{e \mapsto e'}{\mathsf{INL}\ A\ e \mapsto \mathsf{INL}\ A\ e'} \qquad \frac{e \mapsto e'}{\mathsf{INR}\ A\ e \mapsto \mathsf{INR}\ A\ e'}$$

$$\frac{e \mapsto e'}{\mathsf{CASE}\ (e, x.e_1, y.e_2) \mapsto (e', x.e_1, y.e_2)}$$

$$\frac{e\ \mathsf{val}}{\mathsf{CASE}\ (\mathsf{INL}t\ \ e, x.e_1, y.e_2) \mapsto e_1[x := e]}$$

$$\frac{e\ \mathsf{val}}{\mathsf{CASE}\ (\ \mathsf{INR}\ t\ \ e, x.e_1, y.e_2) \mapsto e_2[y := e]}$$

  Estas reglas describen como evaluar los términos introducidos por los tipos variante, INL A e es un valor hasta que e sea un valor, análogamente con INR A e y ABORT A e. En un CASE se debe evaluar la guardia hasta que sea un valor del cual dependerá que sustitución se llevará a cabo.

- Recursión

$$\frac{}{\mathsf{REC}\ x\ \ A\ \ e \mapsto e[x := \mathsf{REC}\ \ x\ \ A\ \ e]}$$

  Esta regla nos indica como se evaluan las funciones recursivas. Esto se logra sustituyendo la función en su mismo cuerpo.

Nótese que el orden de evaluación de los términos es de izquierda a derecha, es decir la evalución es estricta. Se ha preferido la evaluación estricta ante la perezosa ya que la primera es más fácil de implementar.

El mecanismo de paso de parámeros es mediante la substitución, que es básicamente la beta-reducción.

4.3.2. Semántica estática

La semántica estática de un lenguaje consiste de una colección de reglas para imponer restricciones en la formación de programas, las cuales forman usualmente un sistema de tipos. Las frases del lenguaje son clasificadas por tipos, los cuales establecen cómo éllas pueden ser usadas en combinación con otros términos. Superficialmente hablando, el tipo de una expresión de programa predice la forma de su valor; se dice que una expresión de programa esta bien tipificada si esta construida consistentemente con estas predicciones. Por ejemplo, la suma de dos expresiones de tipo entero es de tipo entero; por otro lado la suma de

dos expresiones de tipo booleano expresa que la suma no esta definida sobre booleanos.

A continuación damos una definición inductiva de la semántica estática de minHaskell, para lo cual vamos a usar nuevamente juicios de la forma $\Gamma \vdash e : A$ donde $\Gamma = \{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n\}$ es un conjunto finito de variables, llamado contexto de tipificación, y e es una expresión en sintaxis abstracta.

Definición 4.1. *Las reglas que definen la semántica estática de* minHaskell *son:*

- *Variables y constantes*

$$\frac{}{\Gamma, x : \mathsf{A} \vdash x : \mathsf{A}}\ \textit{(Var)} \qquad \frac{}{\Gamma \vdash \mathsf{INT\ n} : \mathsf{Int}}\ \textit{(Int)} \qquad \frac{}{\Gamma \vdash \mathsf{BOOL\ v} : \mathsf{Bool}}\ \textit{(Bool)}$$

- *Condicionales y operaciones primitivas*

$$\frac{\Gamma \vdash e : \mathsf{Bool} \quad \Gamma \vdash e_1 : \mathsf{A} \quad \Gamma \vdash e_2 : \mathsf{A}}{\Gamma \vdash (\mathsf{IF}\ e\ e_1\ e_2) : \mathsf{A}}\ \textit{(OP)} \qquad \frac{\Gamma \vdash e_1 : \mathsf{A} \ \ldots\ \Gamma \vdash e_n : \mathsf{A}}{\Gamma \vdash \mathsf{OP\ Op}\ (e_1, ..., e_n) : \mathsf{A}}\ \textit{(OP)}$$

- *Abstracciones y aplicaciones*

$$\frac{\Gamma, x : \mathsf{A} \vdash e : \mathsf{B}}{\Gamma \vdash (\mathsf{LAM}\ x\ \ \mathsf{A}\ e) : \ \mathsf{A} \to \mathsf{B}}\ \textit{(I-Abs)} \qquad \frac{\Gamma \vdash e_1 : \mathsf{A} \to \mathsf{B} \quad \Gamma \vdash e_2 : \mathsf{A}}{\Gamma \vdash (\mathsf{AP}\ e\ e_2) : \mathsf{B}}\ \textit{(E-Abs)}$$

- *Recursión*

$$\frac{\Gamma, x : \mathsf{A} \vdash e : \mathsf{A}}{\Gamma \vdash (\mathsf{REC}\ x\, \mathsf{A}\ e) : \ \mathsf{A}}\ \textit{(Fix)}$$

- *Pares*

$$\frac{\Gamma \vdash e : \mathsf{A} \quad \Gamma \vdash e_2 : \mathsf{B}}{\Gamma \vdash \mathsf{PAIR}\ e\ e_2 : \ \mathsf{A} \times \mathsf{B}}\ \textit{(I-Prod)}$$

$$\frac{\Gamma \vdash e : \ \mathsf{A} \times \mathsf{B}}{\Gamma \vdash \mathsf{FST}\, e : \ \mathsf{A}}\ \textit{(E-Prod)} \qquad \frac{\Gamma \vdash e : \ \mathsf{A} \times \mathsf{B}}{\Gamma \vdash \mathsf{SND}\, e : \ \mathsf{B}}\ \textit{(E-Prod)}$$

$$\frac{}{\Gamma \vdash \mathsf{TRIV} : \mathsf{unit}}\ \textit{(triv)}$$

- *Sumas*

$$\frac{\Gamma \vdash e : \ \mathsf{A}}{\Gamma \vdash \mathsf{INL}\ \mathsf{A} + \mathsf{B}\ e : \ \mathsf{A} + \mathsf{B}}\ \textit{(I-Sum)} \qquad \frac{\Gamma \vdash e : \ \mathsf{B}}{\Gamma \vdash \mathsf{INR}\ \mathsf{A} + \mathsf{B}\ e : \ \mathsf{A} + \mathsf{B}}\ \textit{(I-Sum)}$$

$$\frac{\Gamma \vdash e : \mathsf{A} + \mathsf{B} \quad \Gamma, x : \mathsf{A} \vdash e_1 : \mathsf{C} \quad \Gamma, y : \mathsf{B} \vdash e_2 : \mathsf{C}}{\Gamma \vdash (\mathsf{CASE}\, e\ e_1\ e_2) : \mathsf{C}}\ \textit{(E-Sum)}$$

$$\frac{\Gamma \vdash e : \mathsf{void}}{\Gamma \vdash \mathsf{abort}\ \ A\ e\ A}\ \textit{(void)}$$

Como los programas de MINHASKELL están tipificados *a la Church*, esta parte solo se encarga de verificar que los tipos asignados por el programador sean correctos, en contraste con una tipificación *a la Curry* donde tendríamos que implementar un mecanismo de asignación de tipos.

4.4. Seguridad del lenguaje

Que un lenguaje sea seguro de manera informal significa que ciertos errores no ocurren durante la ejecución de un programa, como que una constante booleana sea sumada con un entero. La seguridad de un lenguaje expresa la coherencia entre la semántica estática y la semántica dinámica. La semántica estática puede ser vista como una predicción de la forma que tendrá el valor de una expresión y así garantizar que la semántica dinámica está bien definida. Consecuentemente, la evaluación no puede quedarse en un estado para el cual no hay una transición posible, dicho en otras palabras no hay errores por instrucciones ilegales en tiempo de ejecución. La seguridad de un lenguaje se demuestra al establecer que cada paso de transición preserva el tipo, y mostrando que los estados de tipificación están bien definidos. Esto se expresa de manera formal en las siguientes proposiciones.

Proposición 4.1 (Preservación)**.** *Si* $\Gamma \vdash e : \mathsf{T}$ *y* $e \mapsto e'$ *entonces* $\Gamma \vdash e' : \mathsf{T}$

Demostración. La demostración es por inducción sobre la relación $e \mapsto e'$. □

Proposición 4.2 (Progreso)**.** *Si* $\Gamma \vdash e : \mathsf{T}$ *entonces e es un valor o existe* e' *tal que* $e \mapsto e'$*.*

Demostración. La Demostración es por inducción sobre las reglas de tipificación. □

La preservación asegura que los pasos de evaluación preservan los tipos; la propiedad de progreso establece que expresiones bien tipificadas o son valores, en cuyo caso su evaluación termina con éxito o bien pueden seguir evaluandose. Para mayor información sobre las demostraciones de estas proposiciones consúltese [3].

4.4.1. Terminación

La propiedad de terminación o normalización fuerte es válida para $\lambda^{\rightarrow}$. Esta es la instancia más simple de una técnica importante en el estudio de lenguajes

de programación conocida como *terminación basada en tipos* donde la semántica estática garantiza que los programas terminarán.

Es importante mencionar que MinHaskell admite funciones recursivas ya que corresponde escencialmente al lenguaje PCF (Programming Computable Functions) para mayor información sobre PCF véase [19], que integra funciones y números naturales usando recursión general como medio para definir expresiones que hacen referencia a sí mismas. Los programas en PCF podrían no terminar durante el proceso de evaluación. En este caso es el programador quien debe garantizar que su programa termina, ya que el sistema de tipos no garantiza la terminación de todo programa. Por lo tanto en minHaskell sin la regla para recursión general

$$\frac{\Gamma, x : A \vdash e : A}{\Gamma \vdash (\mathsf{REC}\ x\ A\ e) : A}\ (Fix)$$

Se cumple la propiedad de terminación, no en caso contrario, por ejemplo:

Ejemplo 4.3. *No terminación de todos los programas en* MinHaskell.

$$(\mathsf{REC}\ x\ A\ x) \to x[x := (\mathsf{REC}\ x\ A\ x)] \to (\mathsf{REC}\ x\ A\ x) \ldots$$

Agregar la regla (Fix) permite definir un mayor rango de funciones, al costo de admitir ciclos infinitos es decir, secuencias infinitas de reducción o evaluación en la semántica dinámica. Se conservan las propiedades de progreso y conservación en minHaskell con la regla (Fix).

4.5. Rompiendo el isomorfismo

Hasta ahora hemos descrito un lenguaje de programación cuyo sistema de tipos es escencialmente el presentado en la sección 2.5 salvo la regla que modela recursión, la cual nos permite definir funciones con nombre del estilo

$$\mathsf{fun}\ \ factorial\ (x : \mathsf{Int}) : \mathsf{Int} = \mathsf{if}\ \ x == 1\ \ then\ \ 1\ \ else\ \ x * factorial\ \ (x - 1)$$

La importancia de nombrar funciones no es simplemente una característica que facilita el entendimiento de un programa, sino que proporciona un mecanismo que nos permite definir funciones sin el cual sería imposible definir. Considérese el siguiente ejemplo:

$$2^0 = 1$$
$$2^{(n+1)} = 2 * 2^n \ \ n > 0$$

Esta es una definición recursiva de la función exponencial 2^n, para definir su valor en $n + 1$ hace uso de su valor en n. Nótese que con funciones anónimas (sin nombre) sería imposible definir tal función, un itento sería $\lambda n.2^n$, pero evidentemente esta definición no sirve. Si nombramos la defínción se convierte en:

$$potd\ 0 = 1$$
$$potd\ (n+1) = 2 * (potd\ n)\ \ n > 0$$

La cual proporciona una implementación de la función exponenciación 2^n. No incluir a (Fix) en minHaskell daría como resultado un lenguaje poco expresivo. En general un lenguaje funcional que no implemente recursión es un lenguaje que no sirve para nada.

Por otro lado, resulta de interés saber bajo el *isomorfismo de Curry-Howard* a que corresponde el mecanismo de recursión en la lógica. La derivación correspondiente es la siguiente:

$$\frac{\Gamma, x :\ A \vdash e :\ A}{\Gamma \vdash (\mathsf{REC}\ x\ A\ e) :\ A} \quad \rightarrow \quad \frac{\Gamma^\star, A^\star \vdash A^\star}{\Gamma^\star \vdash A^\star}$$

Esta regla no introduce ni elimina ningún conectivo de LPI, de hecho añadir la derivación correspondiente a nuestro sistema DN provocaría la trivialización del sistema, puesto que esta regla establece que hay que suponer lo que se quiere probar. Por lo que si queremos mantener nuestro sistema DN consistente no debemos agregar tal derivación. Esto provoca que el *isomorfismo de Curry-Howard* presentado en este trabajo se rompa y no se pueda extender a sistemas que modelan recursión con la regla (Fix). Sin embargo esto no quiere decir que no haya una interpretación del lado de la lógica para la recursión, de hecho ésta sería que la autoreferencia no puede ser permitida como evidencia para la verdad de una proposición.

4.6. Implementación

Procedemos a dar una breve descripción sobre algunos detalles importantes de la implementación de minHaskell.

4.6.1. Análisis léxico

El análisis léxico consiste en leer la secuencia de carácteres del programa fuente, caracter a caracter, agruparlos y formar los componentes léxicos o tokens que representan a las palabras reservadas, identificadores asociados a variables y nombres de funciones, tipos definidos por el usuario, operadores, símbolos especiales, y constantes numéricas. El resultado del análisis léxico es pasado al analizador sintáctico. El proceso de transformar un programa, escrito en sintaxis concreta, a una lista de tokens facilita la labor del analizador sintáctico por lo que es indispensable realizarlo. En el archivo Lexer.lhs se definen los siguientes tipos de datos:

- Token. Define los tokens a los que serán traducidos los programas escritos en sintaxis concreta.

- Operaciones. Define los lexemas para los símbolos de las operaciones primitivas válidas, los cuales son: suma (+), multiplicación (∗), igual que (==), menor que (<) y negación (!).
- Pres. Define los lexemas para las palabras reservadas de minHaskell. Las cuales son: let, in, end, fun, else, then, bool, int, true,false, case, of, fst, snd, inr, inl, triv, abort, unit, void.

La función principal es

```
lexer :: [Char] -> [Token]
```

Que transforma un programa de sintaxis concreta a una lista de tokens, que representan al programa. También se encarga de reconocer a los símbolos especiales los cuales son:

```
),(, +,-,  *,->,  =,==,  $, *  ,&, : ,<, !, |,=>
```

Ejemplo 4.4. *Ejemplos de la función lexer.*

- ```
  lexer "fun (x:Int) = x"
  ```

  ```
  [RES Fun,PARI,ID "x",DOSPUNTOS,RES Int,PARD,DEF,ID "x"]
  ```
- ```
  lexer "fun x:Int = x + 5"
  ```

  ```
  [RES Fun,ID "x",DOSPUNTOS,RES Int,DEF,ID "x",O MAS,ENTERO 5]
  ```
- ```
  lexer "2+4+t"
  ```

  ```
  [ENTERO 2,O MAS,ENTERO 4,O MAS,ID "t"]
  ```

4.6.2. Análisis sintáctico

El analizador sintáctico se encaraga de procesar la lista de tokens generada por el analizador léxico transformandola al correspondiente árbol de sintaxis abastracta. El analizador sintáctico de minHaskell se define en el archivo Parser.lhs; El tipo de dato que define la sintaxis abstracta utilizada por el analizador sintáctico se encuentra definido en el archivo Data.lhs. En este archivo se definen los siguiente tipos de datos:

- Exp. Define la sintaxis abstracta presentada en la sección 4.2.
- DBExp. Define la sintaxis abstracta, usando índices de *de Bruijn*.
- Tipo. Define los tipos posibles de los programas.

La función principal de Lexer.lhs es

```
parse :: [Token] ->  Exp
```
```

la cual transforma una lista de tokens en un árbol de sintaxis abstracta.

Ejemplo 4.5. *Uso de la función parse.*

```
Main> parse (lexer "g f 0" )

AP (AP (VAR "g") (VAR "f")) (INT 0)
Main> parse (lexer "let x:Int = 2 in x+2 end" )

AP (LAM "x" TINT (OP Mas [VAR "x",INT 2])) (INT 2)
```

Es importante destacar que nuestro analizador sintáctico se basa en el analizador sintáctico de MinML, escrito en ML, pero nuestro código es mucho más sencillo gracias a la evaluación peresoza de Haskell.

4.6.3. Substitución

Como mencionamos en el capítulo 2, el mecanismo de paso de parámetros en minHaskell es escencialmente la β-reducción, por lo cual en esta sección describimos a grandes rasgos como implementar tal mecanismo. El procedimiento de substitución del cálculo lambda es sencilla, su implementación usando nombres de variables de manera explícita no. Por lo que resulta conveniente tener una representación de éllas que faciliten la implementación de la substitución. Para lograr este objetivo usaremos una representación *anónima* de las funciones, donde el nombre de variables no aparecerá de manera explícita, si no que usaremos índices para representarlas, estos índices se conocen como índices de *de Bruijn*, los cuales describimos a continuación.

Índices de deBruijn

La idea es representar una variable ligada con el número de lambdas que es necesario *saltar* hasta encontrar la lambda que la liga. Veamos unos ejemplos:

$$\lambda.0 \text{ representaría } \lambda x.x$$
$$\lambda\ .0(\lambda.01) \text{ representaría } \lambda z.z(\lambda y.yz)$$

La representación con índices de *de Bruijn* de un término cerrado (sin variables libres) es única, no así si el término tiene variables libres. Por ejemplo si tenemos $\lambda z.zx(\lambda y.zxy)$ su representación anónima (usando índices de de Bruijn) podría ser $\lambda.012$, ésto si cambiamos a x por 1 e y por 2, pero nótese que la elección de los valores de x e y fué arbitraria pudiendo ser de otra manera por ejemplo x por 2 e y por 4, dando como resultado $\lambda.024$. Para poder representar términos con varibles libres de manera anónima es necesario declarar que índices representarán a tales variables mediante un *contexto de índices*. Por ejemplo si definimos $x \mapsto 4$, $y \mapsto 5$ entonces $\lambda z.zx(\lambda y.zxy)$ se convierte en $\lambda.045$. Para poder asignar índices de *de Bruijn* de manera determinista a las variables libres

tomamos la siguiente convención: Dado un término t asignar a cada presencia de una variable libre empezando desde la derecha el menor índice mayor o igual al mínimo número de lambdas necesarios para liberar a la variable de todos los alcances dentro de los que se encuentre, y que no haya sido utilizado hasta ese momento para nombrar variables libres.

Ejemplo 4.6. *Veamos unos ejemplos.*

$$\lambda z.zxy \text{ se representa con } \lambda.021$$
$$\lambda z.zx(\lambda y.zxy) \text{ se representa con } \lambda.01(\lambda.120)$$

En el primer caso para y basta con saltar la única lambda visible, por lo que se representa con un 1, y la x con un 2. En el segundo ejemplo la segunda presencia de x se representa con 2 pues estaba bajo el alcance de dos lambdas presentes. Por otro lado la primera presencia de x se representa con un 1, dado que se necesita sólo un salto para liberarla y el 1 no ha sido usado para variables libres. La asignación de índices de de Bruijn no es una sustitución textual, de hecho una misma variable, ya sea libre o ligada, puede recibir distintos índices como el último ejemplo lo muestra.

La representación anónima no es amigable para las personas, pero su aplicación principal es en la implementación de la substitución del cálculo lambda. Para la formalización de los índices de *de Bruijn*, contextos de índices, etc consúltese [3].

La definición de las funciones que se encargan de tomar un término de minHaskell y devolver su representación anónima, así como el procedimiento inverso se encuentran implementadas en DeBruijn.lhs. Las cuales describimos a continuación.

- `quitaN :: [[Char]] -> Exp -> DBExp`

  La función quita nombres recibe un programa de minHaskell, con nombres de las variables, y devuelve su representación anónima.

- `ponN :: [[Char]] -> DBExp -> Exp`

  La función pon nombres toma un programa de minHaskell, en su representación anónima, y devuelve su representación con nombres.

Nótese que es necesario definir un nuevo tipo de datos que represente este cambio en la sintaxis abstracta, el cual se define en el archivo defData.lhs, como DBExp. Para más información véase [3].

Ejemplo 4.7. *Índices de de bruijn*

- ```
  quitaN []  (LAM VAR "x" TINT (VAR "x"))

  DLAM TINT (DVAR 0)
  ```
  ```

- `quitaN ["z", "x", "y"]  (AP (VAR "x") (VAR "y"))`

  `DAP (DVAR 1) (DVAR 0)`

Una ves que tenemos los términos de minHaskell en su forma anónima explicaremos la idea seguida para implementar la operación de substitución. Se debe tener cuidado con los índices de las variables libres cuando una substitución opera en el cuerpo de una abstracción como $(\lambda.2)[1 := s]$ el contexto interior tiene una variable más que la original (la variable que estaba ligada por la abstracción) de forma que es necesario incrementar los índices de las variables libres en s para que se sigan refiriendo a las mismas variables que antes de la substitución. Por ejemplo si el término $2(\lambda.0)$ en el contexto $z \mapsto 2$, es decir $z(\lambda w.w)$ se va a substituir dentro de una abstracción es necesario incrementar el índice 2 a 3 pero no el índice 0. Considérese la reducción:

$$(\lambda x.(\lambda y.xy)zx)\ (\lambda w.vw) \rightarrow_\beta (\lambda y.(\lambda w.vw)y)z(\lambda w.v.w)$$

El redex se representa con $(\lambda\ .(\lambda\ .10)10)\ (\lambda.\ 20)$ mientras que el reducto debe representarse con

$$\lambda\ .(\lambda\ .30)\ 0(\lambda\ .20)$$

Obsérvese que:

- Las variables libres no afectadas por la substitución ven su índice decrementado, reflejando el hecho de que una lambda desaparece.
- Las variables libres del término por el que se sustituye una varible se ajustan de acuerdo a su posición en el término, es decir de acuerdo a si quedan más o menos lambdas.

Para realizar la implementación de las substitución se requiere de una función auxiliar que se encargue de incrementar o decrementar los índices necesarios, acontinuación definimos tal función.

Definición 4.2. *La función auxiliar shift se define como sigue:*

$\mathsf{shift}\ (d, c, k) = if\ k < c\ then\ k\ else\ k + d$

$\mathsf{shift}\ (d, c, \lambda.t) = \lambda.\mathsf{shift}\ (\mathsf{d}, \mathsf{c} + 1, , \mathsf{t})$

$\mathsf{shift}\ (d, c, rs) = \mathsf{shift}(d, c, r)\ \mathsf{shift}(d, c, t)$

Una ves definida la función shift podemos definir la función que se encarga de la substitución.

Definición 4.3. *La substitución en términos anónimos a[j:=s] se define como sigue:*

$n[j := s] = if\ n = j\ then\ s\ else\ n$

$(\lambda.t)[j := s] = \lambda.t[j + 1 := \mathsf{shift}(1, 0, s)]$

$(tr)[j := s] = t[j := s]r[j := s]$

La implementación de estas funciones se encuentran en el archivo Eval.lhs

- `shift :: (Int,Int,DBExp) -> DBExp`
  Función auxiliar para realizar los desplzamientos en los índices de de Bruijn necesarios.
- `sustituye :: DBExp -> Int -> DBExp -> DBExp`
  Función que realiza la substitución en términos anónimos.

Ejemplo 4.8. *Ejemplos de la función sustituye.*

- ```
  sustituye (DLAM TINT (DVAR 0)) (-1) (DINT 1)

  DLAM Entero (DINT 1)
  ```
- ```
  sustituye (DLAM TINT (DOP  Mas  [DVAR 0,DINT 2])) (-1) (DINT 4)

  DLAM TINT (DOP  MAS  [DINT 4,DINT 2])
  ```

4.6.4. Verificación de tipos

La semántica estática de minHaskell descrita en la sección 4.3.2 se encarga de verificar que el tipo asignado, por el usuario, a un programa sea correcto. Implementamos la función tipo que sigue la siguiente idea: tipo Γ *expresion* = A si y solo si $\Gamma \vdash e : A$.

El archivo donde se implementa la semántica estática de minHaskell es Typing.lhs. cuya función principal es

```
tipo :: [Tipo] -> DBExp -> Tipo
```

la cual recibe un contexto de tipificación y una expresión en su forma anónima y devuelve el tipo de la expresión.

Ejemplo 4.9.

```
tipo [] (DAP (DLAM TINT (DOP Por [DOP Mas [DVAR 0,DINT 4],DINT 34])) (DINT 4))

Entero
tipo [] (DAP (DLAM TINT (DAP (DLAM TINT (DOP Mas [DVAR 1,DVAR 0])) (DINT 44))) (DINT 23))

Entero
```

4.6.5. Evaluación

El proceso de evaluación de los programas en MinHaskell sigue la siguiente idea: *eval* $e = v$ syss $e \rightarrow^* v$, v es un valor. La semántica dinámica de minHaskell se encuentra implementada en el archivo Eval.lhs. Cuya función principal es eval :: DBExp -> DBExp. Esta función recibe una expresión, en su forma anónima, y devuelve un valor que pueden ser un valor booleano, un valor entero, o una abstracción.

Ejemplo 4.10.

```
eval (DAP (DLAM TINT (DOP Por [DOP Mas [DVAR 0,DINT 4],DINT 34])) (DINT 4))

272
eval DAP (DREC (TINT :>: TINT) (DLAM TINT (DIF (DOP Igual [DVAR 0,DINT 1]) (DINT 1)
(DOP Por [DVAR 0,DAP (DVAR 1) (DOP Menos [DVAR 0,DINT 1])])))) (DINT 5)

120

Main> eval DLAM TINT (DOP Mas [DVAR 0,DINT 1])

DLAM TINT (DOP Mas [DVAR 0,DINT 1])
```

4.7. Ejemplos y ejecución

EL módulo principal esta implementado en el archivo Main.lhs. Si se desea probar el interprete de MINHASKELL, suponiendo se esta en una máquina con sistema operativo Linux, basta cargar el módulo principal Main con el interprete de HASKELL, HUGS 98, haciendo: hugs Main.lhs. Una vez hecho esto, el interprete de HASKELL cargará los módulos necesarios, por lo que se desplegará el prompt: Main, en el cual se deberá ejecutar el siguiente comando: main. Se pedirá introducir el nombre del archivo que contiene la especificación del programa (se proporcionan varios), se desplegará en pantalla el programa leido así como el resultado de evaluarlo.

Ejemplo 4.11. *Probando* MINHASKELL.

```
__   __ __  __  ____   ___      ________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 ________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Main> main
Introduzca el nombre del archivo a leer

programa1
El programa leido es:
"(fun factorial (x:Int):Int = if  x==1 then 1 else x * factorial(x-1)) 5"

El valor final es:
DINT 120::Entero
```

Capítulo 5

Conclusiones

Los sistemas lógicos tienen como uno de sus propósitos representar el razonamiento humano, pero ninguno de ellos hasta ahora ha podido capturar toda la riqueza y complejidad de éste. Sin embargo hay algunas lógicas que por la naturaleza de los principios que las fundamentan resultan útiles para describir con cierta precisión algunos conceptos del razonamiento humano. Por ejemplo la lógica proposicional intuicionista (LPI), que se presentó en el primer capítulo, por su naturaleza constructiva resulta conveniente para el estudio del concepto de computo. En élla la noción de verdad de una proposición, esta ligada con la existencia de una demostración de tal proposición. Esto resulta importante en ciencias de la computación ya que a los cientificos de la computación no sólo nos interesa saber cuando un problema tiene solución, si no que además estamos interesados en construir o implementar tal solución, por lo que un fundamento lógico constructivista de algunas áreas de las ciencias de la computación es más que conveniente. Con esta visión en el capítulo uno se desarrolló la LPI mediante un sistema de deducción natural.

Una introducción a los sistemas de tipos se presentó en el capítulo dos, ya que el impacto que éstos tienen en el diseño de los lenguajes de programación modernos, así como en la disciplina de la programación, cada día es mayor. El fundamento de muchos sistemas de tipos reside en el cálculo lambda, el cual es un sistema de cómputo universal, que además puede ser visto como un prototipo de lenguaje de programación funcional, en donde la noción de cómputo está expresada en la β-reducción. A pesar de que el cálculo lambda puro logra expresar con gran cercanía muchos conceptos de cómputo, como la definición de tipos de datos, resulta demasiado expresivo, por ejemplo la autoaplicación, puede provocar que el proceso de evaluación de un término no finalice nunca. Una manera de evitar estos problemas consiste en introducir tipos obteniendo lo que se conoce como cálculo lambda con tipos simples que se presentó en la sección 2.3. La relación de éste con los lenguajes de programación es más notoria, por ejemplo, en los problemas de decisión (ver sección 2.3.3) que surgen al analizar el juicio: $\Gamma \vdash t : A$, los cuales tienen una interpretación en lenguajes de programación de suma importancia.

El *isomorfismo de Curry Howard* vincula dos formalismos: la deducción natural, debida a Gentzen, y el ćalculo lambda, debido a Church y a Curry. Ambos formalismos surgieron con un propósito muy distinto; Gentzen intentaba expresar el razonamiento matemático de manera clara, mientras que Church intentaba fundamentar a las matemáticas. Los trabajos de ambos fueron públicados con poco tiempo de diferencia entre ellos, sin embargo, tomó décadas para que Howard hiciera una *modesta* observación que relaciona ambos sistemas, esta observación la conocemos hoy como el *isomorfismo de Curry Howard*.

En el capítulo tres se abordó de manera detallada este isomorfismo, el cual establece una correspondencia entre tipos y proposiciones, demostraciones y programas, β-reducción y normalización de demostraciones. Como consecuencia de esta correspondecia obtenemos una representación o codificación líneal de las demostraciones de LPI, ya que si bien los árboles de derivación de la deducción natural son conceptualmente útiles también es cierto que resultan difíciles de manipular, además obtenemos un fundamento lógico para los lenguajes de programación funcionales. Este último punto se desarrolló a lo largo del capítulo cuatro donde presentamos de manera formal un pequeño lenguaje de programación: minHaskell, cuyo sistema de tipos, es decir su semántica estática, coincide con el de PCF, y cuya semántica dinámica, es decir, el mecanismo de evaluación, es esencialmente la β-reducción. La relevancia del *isomorfismo de Curry-Howard* en el diseño de este lenguaje radica en la relación de la semántica estática y la semántica dinámica, nuestro lenguaje es seguro debido a esta relación tal como lo discutimos en la sección 4.4. De esta manera el *isomorfismo de Curry-Howard* permite obtener ventajas prácticas a partir de un formalismo lógico para los lenguajes de programación funcional. Sin embargo, con el propósito de tener un lenguaje de programación más cercano a los lenguajes reales agregamos a minHaskell mecanismos para el manejo de funciones recursivas lo que provoca que el isomorfismo se rompa, como lo mencionamos en la sección 4.4. Esto es consecuencia de que no está permitida la autoreferencia como evidencia para la verdad de una proposición.

La influencia y el alcance de esta correspondencia en los lenguajes de computación sigue siendo estudiado [26]. Por ejemplo es natural preguntarse si el *ismorfismo de Curry-Howard* puede extenderse a la lógica clásica (véase [25]), o bien a otros sistemas como la lógica proposicional de segundo orden [7]. Estos y otros aspectos de esta maravillosa correspondencia representan posibles lineas de trabajo futuro.

Bibliografía

[1] Aczel Peter. *Notes On the simply Typed Lambda Calculus*, Proceedings of the 1997 "Computational Logic" Advanced Study Institute International Summer School at Marktberdorf, 1999.

[2] A.S Troelstra y D. Van Dalen, *Constructivism in mathematics: An introduction*, Studies in logic and the foundations of Mathematics. **vol 121,123** 1988.

[3] Benjamin C. Pierce, Types and Programming Languages, MIT Press, 2002.

[4] D. Prawitz. Natural deduction, a proof-theorical study, Almquist & Wiksell, 1965.

[5] Eduardo Bonelli, *Pruebas y Programas*, Curso de Postgrado-Fac. de Informatica-UNLP, Dept. of Computer Science, Stevens Institute of Thecnology, Hoboken, NJ.

[6] Eugene P. Wigner, *The unreasonable effectiveness of mathematics in the natural sciences*, Communications in Pure and Applied Mathematics, **vol 13**, No 1, 1960.

[7] Favio Ezequiel Miranda Perea. *La lógica proposicional de segundo orden*, `http://matematicas.fciencias.unam.mx/favio/`, 2008.

[8] Favio Ezequiel Miranda Perea. *Notas de clase para el curso de Análisis lógico.* `http://matematicas.fciencias.unam.mx/favio/`, 2004.

[9] Favio Ezequiel Miranda Perea. *Notas de clase para el curso de Lenguajes de programación.* `http://matematicas.fciencias.unam.mx/favio/`, 2007.

[10] Frank Pfenning. *Computation and Deduction.* Cambridge University Press, 2003.

[11] Gentezen. *Investigations into Logical Deduction*, Symbolic Logic, **vol 35**, 1970.

[12] Giuseppe Longo. *The Lambda Calculus: connections to Higher Type Recursion Theory, Proof-Theory, Category Theory.* `ftp://ftp.di.ens.fr/pub/users/longo/PhilosophyAndCognition/church-notes.pdf`.

[13] Henk Barendregt. *The Impact of the Lambda Calculus in Logic and Computer Science*, The Bulletin of Symbolic Logic. **vol 3**, Number 2, 181-215 June 1997.

[14] Henk Barendregt, *Lambda Calculi with Types*, `http://www.cs.ru.nl/~henk/papers.html`.

[15] Henk Barendregt, Erik Barendsen. *Introduction To Lambda Calculus. Revised edition*, `http://www.cs.ru.nl/~henk/papers.html`, December 1998.

[16] Imre Lakatos, Pruebas y refutaciones: La lógica del descubrimiento matemático, Madrid Alianza, 1978.

[17] Jean Gallier, *On the correspondence between Proofs and lambda-Terms*, Cahiers du Centre de Logique, Phillipe DeGroote, Editor, Université Catholique de Louvain, 1965.

[18] Jhon McCarthy. *Recursive Functios of symbolic expressions and their Computation by Machine,Part I.* Massachusetts Institute of Technology, 1960.

[19] John C.Mitchell Foundations for programming Languages, MIT Press, 1996.

[20] Joseph Y.Halpern, Robert Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, Victor Vianu, *On the Unusual Effectiveness of Logic in Computer Science*, The Bulletin of Symbolic Logic, **vol 7**, number 2, 213-236, 2001.

[21] Luca Cardelli. *Type Systems*, Microsoft research, chapter for the CRC Handbook of Computer Science and Enggineering, second edition.

[22] Mike Gordon, *Lectures on Introduction to Functional Programming*, Heycok Lecture Room. 1996.

[23] Morten H.Sorensen, Pawel Urzyczyn. *Lectures On the Curry-Howard Isomorphism*, Studies in Logic and the Foundations of Mathematics, Vol 149, Elsevier 2006.

[24] Peter Salinger. *Lecture Notes On Lambda Calculus*, `http://www.mscs.dal.ca/~selinger/papers/#lambdanotes`. Department of Mathematics ans Statistics, Unervisity of Otawa.

[25] Philip Wadler, *Proofs are Programs:19th Century Logic and 21st Century Computing*. June 2000, updated November 2000.

[26] Robert Harper, *Practical Foundations for Programming Languages*, Carnegie Mellon University, Spring, 2008.

[27] R.W. Hamming, *The unreasonable effectiveness of mathematics*, America Mathematical Monthly, 87:81-90, 1980.

[28] Simon Thompson, Haskell The Craft of Fucntional Programming, Addison Wesley, Second Edition, 1999.

[29] Simon Thompson, Type Theory & Functional Programming. Addison Wesley, 1991.

[30] M.Y. Vardi, *The complexity of relational query languages*. In Proc. 14 th ACM Symp. on Theory of Computing, 1982.

[31] W.A. Howard. The Formulae-As-Types Notion Of Construction, Essays of combinatory Logic, Lambda Calculus and formalism, 1980.

[32] Isabelle a generic proof assistant, `www.cl.cam.ac.uk/research/hvg/Isabelle/`.

[33] The coq proof assistant, `www.coq.inria.fr`.