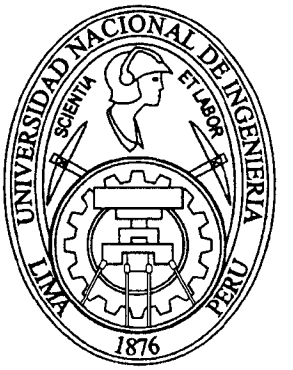
UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE CIENCIAS

ESCUELA PROFESIONAL DE MATEMÁTICA



TESIS PARA OPTAR EL TÍTULO PROFESIONAL DE

LICENCIADO EN MATEMÁTICA

TITULADA:

COMPUTABILIDAD Y DECIDIBILIDAD

PRESENTADA POR:

ORESTES MARTÍN BUENO TANGOA

ASESOR:

OSWALDO JOSÉ VELÁSQUEZ CASTAÑÓN

LIMA-PERÚ

2011

Índice general

Introducción v

1. Preliminares 1
1.1. Relaciones y funciones 1
1.2. Cadenas y lenguajes 3

2. Computabilidad 7
2.1. Algoritmos 7
2.2. λ-cálculo 9
2.2.1. Equivalencias y reducciones de λ-términos 11
2.2.2. Ejecución de un λ-término 14
2.2.3. Aritmética y cálculo booleano en λ-cálculo 15
2.2.4. Recursividad y puntos fijos en λ-términos 19
2.3. Funciones recursivas 20
2.3.1. Funciones recursivas primitivas 20
2.3.2. La función de Ackermann 22
2.3.3. El operador μ, recursividad parcial y recursividad 28
2.3.4. Recursividad de conjuntos 28
2.4. Máquinas de Turing 29
2.4.1. Definición y ejemplos 29
2.4.2. Máquina de Turing de múltiples cintas 37
2.5. La tesis de Church-Turing 41
2.6. Una función no computable: el problema del castor ocupado 42

3. Decidibilidad 45
3.1. Máquinas de decisión y lenguajes decidibles 45
3.2. Máquina de Turing universal y lenguajes no decidibles 48
3.2.1. Codificación de máquinas de Turing 49
3.2.2. La máquina de Turing universal 52
3.2.3. No decidibilidad de lenguajes 54
3.3. Un problema no decidible 58
3.3.1. El problema de la parada 59

4. Conclusiones 62

A. Citas **64**
A.1. El décimo problema de Hilbert 64
A.2. Definición original de la máquina de Turing 64
A.3. Church acerca de la definición de efectivamente computable 65

Introducción

Algoritmos

En el siglo nueve, el califa Abdullāh al-Mā'mūn ibn Harūn estableció en Bagdag una "casa de la sabiduría", una institución de investigación que invitaba a sabios académicos. Entre los primeros académicos se encontraba *Muḥammad ibn Mūsā al-Khwārizmī*. Él escribió varios trabajos, entre los cuales se encuentra "Algoritmi de numero Indorum", nombre en latín pues el original en árabe ya no existe. En este texto, cada hoja comenzaba con la frase "dixit algoritmi" que significa "asi lo dijo al-Khwārizmī". De esta manera, un matemático del siglo VII dió su nombre al concepto central de la ciencia de la computación. A través del trabajo de al-Khwārizmī las técnicas de la aritmética fueron conocidas en Europa como *algorismo*; y el nombre permaneció, como *algoritmo*, para denotar a un procedimiento para resolver un problema en un número finito de pasos.

Sin embargo, algoritmos han existido desde mucho antes de que exista una palabra que los describa. Una vez descubierto un método rutinario para obtener la solución de un problema, no es sorpresa que tal 'receta' fuera compartida a otros para usarse. Los algoritmos no están confinados a la matemática, por ejemplo, los Babilonios los usaban para aplicar leyes, algunas culturas los usaban para predecir el futuro; y actualmente se usan en medicina para diagnosticar enfermedades, y en la cocina, a manera de recetas.

El décimo problema de Hilbert

En 1900, el matemático alemán David Hilbert, en el congreso internacional de matemáticos en París, presentó una lista de 23 problemas, en esa época aún por resolverse. El décimo problema propuesto dictaba lo siguiente[1]

> **10. Determinación de la solubilidad de ecuaciones diofánticas.**
> Dada una ecuación diofántica con cualquier número de incógnitas y con coeficientes numéricos racionales enteros: *Idear un proceso con el cual pueda determinarse, en un número finito de operaciones, si la ecuación es resoluble en números racionales enteros.*

[1] Vea la sección A.1 para el texto original

Pese a que Hilbert no creía en la existencia de problemas insolubles, el décimo problema de Hilbert fue resuelto negativamente gracias al trabajo conjunto de Martin Davis, Julia Robinson y Yuri Matijasevič [Matijasevič, 1970, Davis et al., 1961, Robinson, 1969]. Estos probaron que, de hecho, no existe algoritmo (en el sentido formal dado por Church y Turing) tal que, para cualquier ecuación diofántica, determine si esta tiene soluciones enteras. Problemas de este tipo se denominan *indecidibles*. La existencia de estos problemas muestra un límite a la capacidad humana de resolver problemas mediante un computador, al igual que el teorema de incompletitud de Gödel muestra un límite del razonamiento lógico humano para demostrar afirmaciones.

Computabilidad en la década de 1930

No es posible resolver este problema sin una definición formal de *proceso*. Es así que este problema inició el estudio de la teoría de la computabilidad, desarrollada por Gödel, Church, Turing, Post, Kleene, etc., comenzando en la década de 1930. Dicha teoría esta relacionada fuertemente con la lógica. Los dos logros mas importantes en esta área fueron dados por el descubrimiento de la incompletitud por Gödel [Gödel, 1931] y la definición de computabilidad, dada independientemente por Church [Church, 1936], al definir el *λ-cálculo*, y por Turing [Turing, 1936], al definir las *máquinas de Turing*. El teorema de incompletitud de Gödel dio una respuesta parcial al *segundo* problema de Hilbert[2], mientras que Church y Turing intentaron dar definiciones formales de computabilidad para obtener una función *incomputable*, usando la técnica de Gödel. Más adelante, Gödel y Kleene introdujeron el concepto de *función μ-recursiva* que resulta coincidir con los conceptos previos de computabilidad de Church y Turing.

Objetivo y plan de trabajo

El objetivo de este trabajo es presentar un estudio formal de computabilidad, haciendo énfasis en las máquinas de Turing, así como de decidibilidad. El fundamento teórico usado a lo largo del trabajo estará dado en el capítulo 1. Esto consta de teoría de conjuntos, relaciones y funciones, así como también la teoría de cadenas y lenguajes.

En el capítulo 2 introduciremos las nociones de computabilidad de Turing, Church y Gödel-Kleene, asi como la relación existente con la noción usual de algoritmos. En este caso, enfatizaremos el estudio de las máquinas de Turing. Presentaremos además un ejemplo de función no computable, según las nociones mencionadas. En el capítulo 3, estudiamos la decidibilidad de lenguajes y de problemas, y presentaremos ejemplos de lenguajes y problemas no decidibles.

Finalmente, en el apéndice presentamos transcripciones de los artículos originales de Church y Turing.

[2]Probar que los axiomas de la aritmética son consistentes.

Capítulo 1

Preliminares

En este capítulo daremos las definiciones básicas que usaremos a lo largo de este trabajo.

1.1. Relaciones y funciones

Sean A y B conjuntos no vacíos. Recordemos que el **producto cartesiano** de A y B es el conjunto

$$A \times B = \{(a, b) \ : \ a \in A \text{ y } b \in B\}.$$

Definición 1.1. Una **relación** (o **relación binaria**) de A en B es una terna ordenada $(A, B, \mathcal{R})$ donde $\varnothing \neq \mathcal{R} \subset A \times B$. En este caso, diremos que A es el **conjunto de partida** de la relación, B es el **conjunto de llegada** de la relación y $\mathcal{R}$ es la **regla de correspondencia** de la relación. Cuando no haya ambigüedades respecto a los conjuntos de partida y de llegada, llamaremos a la relación simplemente por el nombre de la regla de correspondencia.

Si $\mathcal{R}$ es una relación de A en B, la **inversa** de $\mathcal{R}$ es una relación $\mathcal{R}^{-1}$ de B en A, definida como

$$\mathcal{R}^{-1} = \{(b, a) \in B \times A \ : \ (a, b) \in \mathcal{R}\}.$$

Definición 1.2. Sea $\mathcal{R}$ una relación de A en B. El **dominio** de $\mathcal{R}$ es el conjunto

$$\text{dom}(\mathcal{R}) = \{a \in A \ : \ \text{existe } b \in B \text{ tal que } (a, b) \in \mathcal{R}\}.$$

El **rango** de $\mathcal{R}$ es el conjunto

$$\text{ran}(\mathcal{R}) = \{b \in B \ : \ \text{existe } a \in A \text{ tal que } (a, b) \in \mathcal{R}\}.$$

Para $a \in \mathrm{dom}(\mathcal{R})$, la **imagen** de a bajo $\mathcal{R}$ es el conjunto

$$\mathcal{R}(a) = \{b \in B \ : \ (a,b) \in \mathcal{R}\}.$$

Finalmente, dado $b \in \mathrm{ran}(\mathcal{R})$, la **imagen inversa** de b bajo $\mathcal{R}$ es el conjunto

$$\mathcal{R}^{-1}(b) = \{a \in A \ : \ (a,b) \in \mathcal{R}\}.$$

Definición 1.3. Sean $\mathcal{R}$ y $\mathcal{S}$ relaciones de A en B y de B en C, respectivamente. Definimos la **composición** de $\mathcal{R}$ y $\mathcal{S}$, $\mathcal{S} \circ \mathcal{R}$, como la relación de A en C dada por

$$\mathcal{S} \circ \mathcal{R} = \{(a,c) \in A \times C \ : \ \text{existe } b \in B \text{ tal que } (a,b) \in \mathcal{R} \text{ y } (b,c) \in \mathcal{S}\}.$$

En este caso,

$$\mathrm{dom}(\mathcal{S} \circ \mathcal{R}) = \{a \in A \ : \ a \in \mathrm{dom}(\mathcal{R}) \text{ y } \mathcal{R}(a) \cap \mathrm{dom}(\mathcal{S}) \neq \emptyset\},$$
$$\mathrm{ran}(\mathcal{S} \circ \mathcal{R}) = \{c \in C \ : \ c \in \mathrm{ran}(\mathcal{R}) \text{ y } \mathcal{S}^{-1}(c) \cap \mathrm{ran}(\mathcal{R}) \neq \emptyset\}.$$

Definición 1.4. Sea $\mathcal{R}$ una relación de A en B, y $C \subset A$. Definimos la **restricción** de $\mathcal{R}$ a C, $\mathcal{R}|_C$, como la relación de C en B dada por

$$\mathcal{R}|_C = \{(a,b) \in C \times B \ : \ (a,b) \in \mathcal{R}\}.$$

Definición 1.5. Diremos que una relación $\mathcal{R}$ de A en B es

1. una **función**, si para cada $a \in \mathrm{dom}(\mathcal{R})$, $\mathcal{R}(a)$ es unitario;
2. **inyectiva**, si para cada $b \in \mathrm{ran}(\mathcal{R})$, $\mathcal{R}^{-1}(b)$ es unitario;
3. **totalmente definida**, si $\mathrm{dom}(\mathcal{R}) = A$;
4. **sobreyectiva**, si $\mathrm{ran}(\mathcal{R}) = B$;
5. **bijectiva**, si es una función, totalmente definida, inyectiva y sobreyectiva.

Además, diremos que $\mathcal{R}$ es **parcialmente definida**, cuando no es totalmente definida.

En el caso de trabajar con funciones, es usual a la regla de correspondencia como **gráfico**, mientras que el término *regla de correspondencia* es usado para denotar a la *fórmula* que define la función.

En el caso que $\mathcal{R}$ sea una relación de A en A, diremos simplemente que $\mathcal{R}$ es una relación en A. En el caso que $\mathcal{R}$ sea una función de A en B, denotaremos $\mathcal{R} : A \to B$. Esta notación no dice nada respecto al dominio de $\mathcal{R}$.

Definición 1.6. Diremos que una relación $\mathcal{R}$ en A es:

1. **reflexiva**, si para todo $x \in A$, $(x, x) \in \mathcal{R}$;
2. **simétrica**, si para todo $(x, y) \in \mathcal{R}$, $(y, x) \in \mathcal{R}$;
3. **antisimétrica**, si $(x, y) \in \mathcal{R}$ y $(y, x) \in \mathcal{R}$ implica $x = y$;
4. **transitiva**, si para todo $(x, y), (y, z) \in \mathcal{R}$, $(x, z) \in \mathcal{R}$;
5. una **relación de equivalencia**, si es reflexiva, simétrica y transitiva;
6. un **orden parcial**, si es reflexiva, antisimétrica y transitiva;
7. de **tricotomía**, si $A \times A$ es unión disjunta de $\mathcal{R}$, $\mathcal{R}^{-1}$ y $\{(x, x) : x \in A\}$.

1.2. Cadenas y lenguajes

Esta sección fue obtenida de [Sipser, 1996, §0.2] y [Hopcroft et al., 2001, §1.5]. Las cadenas de caracteres y los lenguajes serán fundamentales en los capítulos que siguen. En esta sección llamaremos **alfabeto** a cualquier conjunto no vacío y a cualquier elemento de un alfabeto lo llamaremos **caracter**. Ejemplos de alfabetos que consideraremos durante este trabajo serán

$$\Gamma_1 = \{0, 1\},$$
$$\Gamma_2 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}, \mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\},$$
$$\Gamma_3 = \{\alpha, \beta, \gamma\}.$$

A Γ_2, nuestro alfabeto usual, lo denotaremos como A_β.

Queremos definir ahora una "palabra" sobre un alfabeto dado. Una definición natural sería definir una palabra como una upla ordenada de caracteres, sin embargo, al hacer esto no consideraríamos los casos de palabras de "longitud cero" y "longitud uno". Así tendremos especial cuidado al definir palabras de longitud menor que dos. Recordemos que el concepto de upla ordenada puede ser definido inductivamente como: $(a, b) := \{a, \{a, b\}\}$, para 2-uplas; y $(a_1, \ldots, a_n) := ((a_1, \ldots, a_{n-1}), a_n)$, para n-uplas, con $n \geqslant 3$.

Definición 1.7. Sea $r \geqslant 0$. Una **cadena** (o **palabra**) de **longitud** r sobre un alfabeto Γ, es:

1. una upla ordenada $(u_1, \ldots, u_r)$, cuando $r \geqslant 2$,
2. un conjunto unitario $\{u\} \subset \Gamma$, en el caso que $r = 1$,
3. el conjunto vacío, en el caso que $r = 0$.

Denotaremos $(u_1, \ldots, u_r)$ como $u_1 \cdots u_r$, en caso $r \geqslant 2$; y $\{u\}$ como u, en el caso de palabras de longitud uno. Llamaremos a la cadena de longitud cero como **cadena vacía** y la denotaremos por ϵ. Finalmente, denotamos el conjunto de palabras de longitud r por Γ^r.

Ejemplo 1.8. Sea $\Gamma_1 = \{0,1,2,3,4,5,6,7,8,9\}$; ejemplos de cadenas sobre Γ_1 son: `0010`, `10123`, `1`, `11223344`, etc. Si $\Gamma_2 = \{\mathtt{a},\mathtt{h},\mathtt{l},\mathtt{o}\}$, ejemplos de cadenas sobre Γ_2 son: `hola`, `halosa`, `alohola`, `lola`, etc. Observe que usamos la notación mencionada en la definición, es decir, escribimos 0010 en lugar de $(0,0,1,0)$ por ejemplo.

Dado un alfabeto Γ, denotaremos al conjunto de cadenas sobre Γ como Γ^*, es decir,

$$\Gamma^* = \bigcup_{r\in\mathbb{N}\cup\{0\}} \Gamma^r$$

Dada una cadena $w \in \Gamma^*$, la longitud de w será denotada como $|w|$. Además, dado $i \in \{1,\ldots,|w|\}$ denotaremos por w_i al i-ésimo elemento de w.

Ejemplo 1.9. Sea $\Gamma = \{\mathtt{a},\mathtt{h},\mathtt{l},\mathtt{o}\}$ y $w =$ `aloha`. Entonces $|w| = 5$, $w_1 = w_5 = \mathtt{a}$, $w_2 = \mathtt{l}$, $w_3 = \mathtt{o}$ y $w_4 = \mathtt{h}$.

Definición 1.10. Dados $m, b \in \mathbb{N}$, denotaremos por $m_{(b)}$ a la cadena sobre $I_b = \{0,\ldots,b-1\}$ que representa m en base b. Es decir,

$$m_{(b)} = a_1\cdots a_r \in I_b^* \text{ si, y solamente si } m = a_1 b^{r-1} + \cdots + a_{r-1}b + a_r.$$

Definición 1.11. Sea Γ un alfabeto y $\leqslant$ un orden total sobre Γ. Sean dos cadenas $v = v_1\cdots v_r$ y $w = w_1\cdots w_s$ en Γ^*. Denotaremos $v <_{\text{lex}} w$ si, y solamente si $r < s$ o

$$r = s \text{ y existe } i \in \{1,\ldots,r\} \text{ tal que } v_i < w_i \text{ y } v_j = w_j \text{ para todo } j < i.$$

También denotaremos $v \leqslant_{\text{lex}} w$ si $v <_{\text{lex}} w$ o $v = w$. La relación $\leqslant_{\text{lex}}$ sobre Γ^* se denominará **orden lexicográfico** de Γ^*.

Definición 1.12. Sean $v = v_1\cdots v_r$ y $w = w_1\cdots w_s$ cadenas sobre Γ_1 y Γ_2 respectivamente. La **concatenación** de v y w, denotada por $v \circ w$, o simplemente vw, será la cadena $v_1\cdots v_r w_1\cdots w_s$ (sobre el alfabeto $\Gamma_1 \cup \Gamma_2$).

Es inmediato ver que la cadena vacía ϵ actúa como elemento neutro de $\circ$, es decir $w \circ \epsilon = \epsilon \circ w = w$ para toda cadena w, y además es asociativa, es decir $u \circ (v \circ w) = (u \circ v) \circ w$, para toda terna de cadenas u, v, w.

Definición 1.13. Sea w una cadena sobre Γ. Diremos que x es una **subcadena** de w si existen cadenas u, v tales que $w = uxv$. Diremos que x es un **prefijo** de w si $w = uxv$ con $u = \epsilon$, y diremos que x es un **sufijo** de w si $w = uxv$ con $v = \epsilon$.

Ejemplo 1.14. Sea la cadena $x =$ `lejo` sobre A_β. Entonces x es prefijo de `lejos`, sufijo de `perplejo` y subcadena de estos dos y de `callejon`.

Definición 1.15. Sea Γ un alfabeto y $a \in \Gamma$. Para $n \geqslant 0$, definimos la cadena a^n como

$$a^n = \begin{cases} \epsilon, & \text{si } n = 0, \\ aa^{n-1}, & \text{si } n > 0. \end{cases}$$

Definición 1.16. Sea w una cadena sobre Γ. Definimos la **cadena inversa** w^{-1}, como

$$w^{-1} = \begin{cases} \epsilon, & \text{si } |w| = 0, \\ av^{-1}, & \text{si } |w| = n,\ w = va \text{ con } |v| = n-1. \end{cases}$$

Ejemplo 1.17. Sean las cadenas $x = \texttt{nabos}$, $y = \texttt{somos}$ y $z = \texttt{sopas}$ sobre el alfabeto A_β. Entonces $x^{-1} = \texttt{soban}$, $y^{-1} = \texttt{somos} = y$ y $z^{-1} = \texttt{sapos}$.

De la definición de cadena inversa, tenemos que si v y w son cadenas, entonces se cumple $(vw)^{-1} = w^{-1}v^{-1}$.

Ahora estudiaremos a los conjuntos de cadenas sobre un alfabeto fijado.

Definición 1.18. Un **lenguaje** sobre un alfabeto Γ es un subconjunto de Γ^*.

En adelante, Γ denotará a la vez el alfabeto y el conjunto de cadenas de longitud uno.

Observe que un lenguaje $L \subset \Gamma^*$ puede ser finito o infinito. En el caso que L sea finito, podemos representar L por extensión, es decir, listar todos los elementos de L. En caso L sea infinito, no queda otra que representarlo por comprensión, es decir, escribir

$$L = \{w \in \Gamma^* \ : \ w \text{ cumple cierta propiedad } P\}.$$

Además, como Γ es finito, entonces Γ^* es numerable y por lo tanto cualquier lenguaje L sobre Γ es numerable.

Desde que los lenguajes son conjuntos, tiene sentido aplicarles las operaciones de unión, intersección, diferencia y complemento. En lo que sigue, denotaremos por $\bar{L} = \Gamma^* \backslash L$ al **complemento** de L.

Así como podemos concatenar cadenas, podemos considerar el conjunto de las concatenaciones de cadenas sobre dos lenguajes dados.

Definición 1.19. Sean L_1 y L_2 lenguajes sobre Γ. La **concatenación** de L_1 y L_2 es el lenguaje de las cadenas formadas al concatenar una cadena de L_1 y una cadena de L_2, es decir,

$$L = L_1 \circ L_2 = \{u \circ v \in \Gamma^* \ : \ u \in L_1,\ v \in L_2\}.$$

Una última operación sobre lenguajes es la "estrella de Klein".

Definición 1.20. Sea L un lenguaje sobre Γ. La **estrella de Klein** de L es el lenguaje formado por la

cadena vacía y las cadenas formadas por concatenar una[1] o más cadenas de L, es decir,

$$L^* = \{w \in \Gamma^* \ : \text{ existen } w_1, \ldots, w_r \in L \text{ tales que } w = w_1 \circ \cdots \circ w_r\} \cup \{\epsilon\}$$

[1] Es decir, una sola cadena de L

Capítulo 2

Computabilidad

2.1. Algoritmos

Empezaremos dando nuestra definición de algoritmo.

Definición 2.1. Un **algoritmo** es un conjunto finito de instrucciones que se ejecutan para lograr un objetivo preestablecido.

La definición anterior no es una definición formal desde el punto de vista matemático, pues el concepto de "instrucción" no ha sido debidamente formalizado. Sin embargo, bajo ciertas hipótesis, es posible hacer un tratamiento "casi formal" de los algoritmos.

Estableceremos las siguientes propiedades de los algoritmos:

1. Toda instrucción del algoritmo debe ser lo suficientemente simple como para que cualquier persona pueda ejecutarla (cada instrucción en tiempo constante), y no deben ser ambigua en su ejecución.
2. Todo algoritmo tiene un *estado inicial*, que describe la situación previa a la ejecución del algoritmo.
3. Todo algoritmo tiene un *estado final*, que describe la situación posterior a la ejecución del algoritmo.
4. En todo algoritmo se puede hacer uso de *variables*, que son objetos que pueden almacenar información.
5. Todo algoritmo debe detener su ejecución eventualmente, para cualquier estado inicial.

Al estado inicial y final de un algoritmo se les denomina **entrada** y **salida** del algoritmo, y a las posibles entradas del algoritmo las llamaremos **instancias**. Además, si un conjunto de instrucciones no cumple el ítem 5 de la lista anterior, entonces se le denominará **procedimiento**.

Ahora algunos ejemplos:

Ejemplo 2.2. Podemos considerar las recetas de cocina como algoritmos:

Algoritmo 1 (Algoritmo para hacer limonada).
Estado inicial: Agua en una jarra, limones, azúcar, vaso vacío, cuchara de té, cuchillo, exprimidor.

```
Inicio
1. Verter agua de la jarra al vaso vacío.
2. Partir el limón a la mitad.
3. Exprimir el limón usando el exprimidor.
4. Verter el zumo del limón en el vaso.
5. Echar azucar al gusto.
6. Revolver el contenido del vaso usando la cuchara de té.
Fin
```

Estado final: Vaso de limonada.

Ejemplo 2.3. Considere el problema de *factorización entera*: dado un número natural n compuesto, hallar un entero positivo d que sea divisor de n, distinto de 1 y de n mismo. Este problema se resuelve de manera muy simple con el siguiente algoritmo:

Algoritmo 2 (Fuerza bruta).
Entrada: n número compuesto.

```
Inicio
d := 2
r := n mod d
mientras r <> 0 hacer
    d := d + 1
    r:= n mod d
fin-mientras
Fin
```

Salida: d divisor no trivial de n.

Respecto al ejemplo 2.3, el lector puede encontrar información acerca de la estructura `mientras - hacer - fin-mientras`, entre otras, en [Tenenbaum and Augenstein, 1983].

Pese a que podemos usar algoritmos para realizar prácticamente cualquier tipo de tarea, enfocaremos nuestro trabajo en *computar* funciones naturales. Mas precisamente, tenemos la siguiente definición.

Definición 2.4. Diremos que una función $f : \mathbb{N}^r \to \mathbb{N}$ es **computable** (o **efectivamente calculable**) si existe un algoritmo que, con entrada $\overline{n} \in \mathbb{N}^r$, tenga como salida $f(\overline{n})$.

En matemáticas, es muy usual usar el término *"dada una función f"*, el cual significa que suponemos conocido el conjunto de partida, llegada y regla de correspondencia de f. Esto a su vez significa que

sabemos el valor de f evaluado en cualquier elemento de su dominio. Sin embargo, incluso en el contexto de funciones naturales, esto sucede solo de manera teórica, y no significa que podemos calcular el valor de tal función en cada elemento de su dominio. Para probar esto, mostraremos en la sección 2.6 un ejemplo explícito de función no computable.

Como la definición 2.4 depende de la definición de algoritmo, entonces esta no es estrictamente formal. En la década de 1930, Church, Turing, Kleene, entre otros, intentaron formalizar este concepto. De esto, resultaron tres nociones diferentes de computabilidad: el *λ-cálculo* de Church, las *funciones recursivas* de Kleene y Gödel, y las *máquinas de Turing*.

2.2. λ-cálculo

Una variante del λ-cálculo fue introducido por Church [Church, 1932, Church, 1933] como parte de un artículo sobre fundamentos de matemática. Sin embargo, fue probado por Kleene y Rosser [Kleene and Rosser, 1935] que tal sistema presentaba inconsistencias. Church luego corrigió esto y junto con Klenne presentó el λ-cálculo en la forma que mostramos ahora [Church, 1936, Kleene, 1935].

Consideremos el alfabeto $A = \{\lambda, ., (,), \xi, |\}$. Una **variable** es una cadena de la forma $\xi \cdots \xi|$. Por simplicidad, denotaremos tales palabras por letras en minúsculas, x, y, z, etc.

Definición 2.5. El lenguaje Λ de los **λ-términos**, es el menor lenguaje sobre A tal que

1. contiene todas las variables;
2. si x es una variable y $M \in \Lambda$ entonces $(\lambda x.M) \in \Lambda$;
3. si $M, N \in \Lambda$ entonces $(MN) \in \Lambda$.

A los λ-términos definidos en 2 y 3 se les denomina **abstracciones** y **aplicaciones**, respectivamente. Si $M = (\lambda x.N)$ es una abstracción, llamaremos a la variable x como **parámetro** de M y al λ-término N como **alcance** de M.

Informalmente, la abstracción $\lambda x.M$ se interpreta como la función $x \mapsto M$, y la aplicación (MN) se interpreta como $M(N)$ (M evaluado en N). Estas interpretaciones adquirirán sentido en la próxima sección.

Para simplificar la notación de los λ-términos, estableceremos algunas convenciones:

1. Omitiremos los paréntesis externos de las abstracciones y aplicaciones, siempre que no haya ambigüedad.
2. Las aplicaciones son asociativas por izquierda, es decir, escribiremos MNP en lugar de $((MN)P)$, cuando no haya ambigüedad.
3. Las abstracciones son asociativas por derecha, es decir, escribiremos $\lambda x.\lambda y.M$ en lugar de $\lambda x.(\lambda y.M)$, cuando no haya ambigüedad. En este caso, también escribiremos $\lambda xy.M$.

4. En ausencia de paréntesis, el alcance de λ en una abstracción es el mayor posible. Por ejemplo, escribiremos $\lambda x.xyz$ en lugar de $\lambda x.(xyz)$, sin embargo, no es posible omitir los paréntesis en la expresión $(\lambda x.x)yz$.

Ejemplo 2.6. Presentamos algunos ejemplos de λ-términos y sus interpretaciones:

1. $\lambda x.x \equiv x \mapsto x$, es decir, la función identidad.
2. $\lambda xy.x \equiv \lambda x.\lambda y.x \equiv (x, y) \mapsto x$, la proyección sobre la primera variable.
3. $\lambda xy.y \equiv \lambda x.\lambda y.y \equiv (x, y) \mapsto y$, la proyección sobre la segunda variable.
4. $\lambda xy.xy \equiv \lambda x.\lambda y.(xy) \equiv (x, y) \mapsto x(y)$, una función que dadas dos variables, devuelve la primera evaluada en la segunda.
5. $\lambda x.xx \equiv x \mapsto x(x)$, una función que devuelve su entrada evaluada en si misma.
6. $\lambda x.y \equiv x \mapsto y$, una función constante.
7. $\lambda x.yx \equiv x \mapsto y(x)$, función que evalúa la variable y en su entrada.

Definición 2.7. Sea M un λ-término. Denotemos por $FV(M)$ el **conjunto de variables libres** de M, definido como

1. $FV(x) = \{x\}$.
2. $FV(\lambda x.M) = FV(M) \backslash \{x\}$.
3. $FV(MN) = FV(M) \cup FV(N)$.

Definición 2.8. Sea M un λ-término. Denotemos por $BV(M)$ el **conjunto de variables limitadas** de M, definido como

1. $BV(x) = \varnothing$.
2. $BV(\lambda x.M) = \{x\} \cup BV(M)$.
3. $BV(MN) = BV(M) \cup BV(N)$.

Ejemplo 2.9. Consideremos algunos de los λ-términos del ejemplo 2.6

1. $FV(\lambda x.x) = \varnothing$, $BV(\lambda x.x) = \{x\}$.
2. $FV(\lambda xy.x) = \varnothing$, $BV(\lambda xy.x) = \{x, y\}$.
6. $FV(\lambda x.y) = \{y\}$, $BV(\lambda x.y) = \{x\}$.
7. $FV(\lambda x.yx) = \{y\}$, $BV(\lambda x.yx) = \{x\}$.

Diremos que M es un λ-término **cerrado** si $FV(M) = \varnothing$. Así, los ejemplos 1 y 2 del ejemplo 2.6 son cerrados. Diremos que una variable z es **independiente** del λ-término M, si $z \notin FV(M) \cup BV(M)$. De manera análoga, una variable es independiente de un conjunto finito de λ-términos si es independiente de cada uno de ellos. Como existen infinitas variables entonces siempre existen variables independientes a cualquier conjunto finito de λ-términos.

No siempre los conjuntos FV y BV son disjuntos. Por ejemplo, considere $M = (\lambda x.x)x$. En este caso tenemos $FV(M) = \{x\} = BV(M)$. Observe que en M podemos *renombrar* la variable limitada x por cualquier otra variable sin afectar el *significado* de M. Veremos los detalles de esto en la siguiente sección.

2.2.1. Equivalencias y reducciones de λ-términos

Sean M un λ-término, $x \in BV(M)$ y z una variable. Denotaremos $M\{x \mapsto z\}$ al λ-término tal que toda ocurrencia de la variable limitada x en M es reemplazada por la variable z. Por ejemplo, si $M = (\lambda x.x)x$ entonces $M\{x \mapsto z\} = (\lambda z.z)x$ y no $(\lambda z.z)z$.

Así, estamos en condiciones de definir la *α-equivalencia*. Informalmente hablando, dos λ-términos son α-equivalentes si son iguales salvo renombramiento de sus variables limitadas.

Definición 2.10. La **α-equivalencia** de dos λ-términos M y N, denotada por $M =_\alpha N$ esta definida por las siguientes reglas:

1. $M =_\alpha N$, si M y N son exactamente la misma variable, es decir si $M = N = x$;
2. $M =_\alpha N$, si escribimos $M = M_1M_2$ y $N = N_1N_2$ entonces $M_1 =_\alpha N_1$ y $M_2 =_\alpha N_2$;
3. $M =_\alpha N$, si escribimos $M = \lambda x.M_1$ y $N = \lambda x.N_1$ entonces $M_1 =_\alpha N_1$;
4. $M =_\alpha N$, si escribimos $M = \lambda x.M_1$ y $N = \lambda y.N_1$ entonces $M_1\{x \mapsto z\} =_\alpha N_1\{y \mapsto z\}$, para alguna variable z independiente de M y N.

Es fácil probar que $=_\alpha$ es una relación de equivalencia. Asimismo, es fácil verificar que cualquier α-equivalencia de un λ-término cerrado es cerrada.

Ejemplo 2.11. Los siguientes λ-términos son α-equivalentes:

1. $\lambda x.x =_\alpha \lambda y.y$.
2. $\lambda xy.xyz =_\alpha \lambda ab.abz$.
3. $x(\lambda xz.xy) =_\alpha x(\lambda wz.wy)$.

Hay que tener cierto cuidado al reemplazar variables limitadas de λ-términos, como muestran los siguientes ejemplos.

Ejemplo 2.12. Consideremos $M = \lambda x.yx$. Entonces $M\{x \mapsto y\}$ no es α-equivalente a M pues $M\{x \mapsto y\}$ es un λ-término cerrado, mientras que M no lo es.

Ejemplo 2.13. Considere $M = \lambda z.(\lambda z.zy)$, $N = \lambda z.(\lambda w.wy)$ y $P = \lambda z.(\lambda w.zy)$. Es claro que

$$P\{w \mapsto z\} = N\{w \mapsto z\} = \lambda z.(\lambda z.zy) = M.$$

Se tiene $M =_\alpha N$ pero $M \neq_\alpha P$. En efecto,

$$\begin{aligned} M =_\alpha N &\iff \lambda z.zy =_\alpha \lambda w.wy \\ &\iff \lambda\theta.\theta y =_\alpha \lambda\theta.\theta y \\ &\iff \theta y =_\alpha \theta y, \end{aligned}$$

$$\begin{aligned} M =_\alpha P &\iff \lambda z.zy =_\alpha \lambda w.zy \\ &\iff \lambda\theta.\theta y =_\alpha \lambda\theta.zy \\ &\iff \theta y =_\alpha zy \\ &\iff \theta =_\alpha z. \end{aligned}$$

Así, debemos dar reglas de substitución para λ-términos, de tal manera que se respete la α-equivalencia.

Definición 2.14. Sea N un λ-término y x una variable. Definimos el λ-término $M\{x \mapsto N\}$ inductivamente como sigue:

1. $x\{x \mapsto N\} = N$.
2. $y\{x \mapsto N\} = y$, cuando $y \neq x$ es variable.
3. $(PQ)\{x \mapsto N\} = (P\{x \mapsto N\})(Q\{x \mapsto N\})$.
4. $(\lambda x.P)\{x \mapsto N\} = (\lambda x.P)$.
5. Si $x \notin FV(P)$ o $y \notin FV(N)$ entonces $(\lambda y.P)\{x \mapsto N\} = \lambda y.(P\{x \mapsto N\})$.
6. Si $x \in FV(P)$ y $y \in FV(N)$, entonces $(\lambda y.P)\{x \mapsto N\} = (\lambda z.P\{y \mapsto z\})\{x \mapsto N\}$, donde z es independiente de P y N.

Ejemplo 2.15 (Continuación del ejemplo 2.13). Considere los λ-términos M, N y P dados en el ejemplo 2.13. No es difícil probar que, bajo las reglas dadas en la definición 2.14, tenemos $P\{w \mapsto z\} = P$ y $N\{w \mapsto z\} = N$.

Ejemplo 2.16. Consideremos los λ-términos $M = (\lambda z.yz)$ y $N = z$, y determinemos $M\{y \mapsto N\}$. Como $z \in FV(N) = \{z\}$ y $y \in FV(P)$ entonces, por 6,

$$\begin{aligned}(\lambda z.yz)\{y \mapsto z\} &= (\lambda w.(yz)\{z \mapsto w\})\{y \mapsto z\} \\ &= (\lambda w.yw)\{y \mapsto z\} \\ &= \lambda w.zw\end{aligned}$$

En este caso, para reemplazar y por z en M, tuvimos que cambiar la variable limitada z de M por otra independiente.

Un **redex** (nombre que proviene del inglés *reducible expression*) es un término de la forma

$$(\lambda x.M)N.$$

Un redex representa la evaluación de la función $\lambda x.M$ en el argumento N. Para obtener el resultado, necesitamos realizar las operaciones dadas en M reemplazando N en lugar de la variable limitada x. Esta es la regla principal del λ-cálculo.

Definición 2.17. La **β-reducción** del redex $(\lambda x.M)N$, es el λ-término $M\{x \mapsto N\}$. En este caso, denotaremos

$$(\lambda x.M)N \to_\beta M\{x \mapsto N\}.$$

Un **contexto** es un λ-término $C[*]$ donde $*$ es una variable libre que aparece solo una vez en $C[*]$. Si M es un λ-término denotaremos por $C[M]$ al λ-término que resulta de reemplazar $*$ por M.

Definición 2.18. Definimos la relación de **β-reducción** como todos los pares de la forma

$$\big((\lambda x.M)N, M\{x \mapsto N\}\big) \in \Delta \times \Delta,$$

generados por la regla de β-reducción, junto con los pares de la forma $\big(C[P], C[P']\big)$ donde P es un redex y P' es tal que $P \to_\beta P'$. Si el par (Q, Q') esta en esta relación, escribiremos $Q \to_\beta Q'$. Finalmente, escribiremos $Q \to_\beta^* Q'$ si existen λ-términos $Q_0, Q_1 \ldots, Q_n$ tales que

$$Q = Q_0 \to_\beta Q_1 \to_\beta \cdots \to_\beta Q_n = Q'.$$

A esto último se le llama **sucesión de β-reducciones**.

Ejemplo 2.19.

1. Veamos como aplicar las nociones de contexto y β-reducción para reducir el λ-término

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

Sea $M = \lambda y.xy$ y $N = \lambda x.x$. Entonces

$$(\lambda x.M)N \rightarrow_\beta M\{x \rightarrow N\} = \lambda y.(\lambda x.x)y.$$

Observe que en esta última expresión tenemos el redex $(\lambda x.x)y$, el cual se reduce

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

Luego, usando el contexto $C[*] = \lambda y.*$ tenemos que

$$C[(\lambda x.x)y] \rightarrow_\beta C[y],$$

es decir $\lambda y.(\lambda x.x)y \rightarrow_\beta \lambda y.y$.

2. Consideremos ahora el λ-término

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

Consideremos el contexto $C[*] = *u$. Como el término $(\lambda x.\lambda y.x)(\lambda z.z)$ es un redex entonces podemos β-reducirlo como

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

que implica

$$(\lambda x.\lambda y.x)(\lambda z.z)u = C[(\lambda x.\lambda y.x)(\lambda z.z)] \rightarrow_\beta C[\lambda_y.(\lambda z.z)] = \lambda_y.(\lambda z.z)u.$$

Este último término es un redex que se β-reduce a $\lambda z.z$, luego

$$(\lambda x.\lambda y.x)(\lambda z.z)u \rightarrow_\beta^* \lambda z.z.$$

Queda claro de estos últimos ejemplos que el concepto de *contexto* sirve para formalizar el hecho de reducir redexes en el interior de un λ-término.

2.2.2. Ejecución de un λ-término

Hasta el momento no es inmediato ver a los λ-términos como *programas* que pueden ejecutarse. Aunque puede interpretarse una β-reducción como la ejecución de una instrucción, no es claro aun en que momento dejar de reducir, y cual seria la noción equivalente de *salida* de un programa. En esta sección llenaremos estos vacíos.

Comenzaremos definiendo una noción de salida.

Definición 2.20. Se dice que un λ-término está en **forma normal** si no contiene ningún redex. En este caso, diremos que un λ-término es **normalizable** si se puede β-reducir a un λ-término en forma normal.

Así, dado un λ-término, la noción de ejecución seria reducirlo hasta obtener un término en forma normal. Por ejemplo, el λ-término $\lambda xyz.((\lambda a.x(\lambda b.ab))xy)$ es normalizable, pues

$$\lambda xyz.((\lambda a.x(\lambda b.ab))yz) \to_\beta \lambda xyz.(x(\lambda b.yb)z),$$

y el último λ-término esta en forma normal.

Existen, sin embargo, λ-términos que no son normalizables. Consideremos por ejemplo el redex

$$\Omega = (\lambda x.xx)(\lambda x.xx).$$

Al intentar reducir Ω, obtenemos

$$(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx).$$

Es decir, cualquier sucesión de β-reducciones de Ω es infinita.

La unicidad de la forma normal de un λ-término normalizable se desprende de la siguiente proposición, cuya prueba puede ser encontrada en [Church and Rosser, 1936].

Proposición 2.21. *La β-reducción de λ-términos es **confluente**, es decir, para todo λ-término M, y todo par de β-reducciones de M, M_1 y M_2, existe M' tal que $M_1 \to_\beta^* M'$ y $M_2 \to_\beta^* M'$.*

Corolario 2.22. *Todo λ-término normalizable se β-reduce a una única forma normal.*

Demostración: Es consecuencia de la confluencia de $\to_\beta$ y de observar que si M está en forma normal y $M \to_\beta^* M'$ entonces $M = M'$. □

Esto justifica la noción de ejecución dada a la β-reducción de λ-términos normalizables, pues la *salida* está bien definida.

2.2.3. Aritmética y cálculo booleano en λ-cálculo

La sintaxis del λ-cálculo es muy simple: un λ-término puede ser una variable o una abstracción o una aplicación. No hemos dado sintaxis para representar números o variables booleanas. De hecho, esto no es necesario como veremos en esta sección.

Definición 2.23. Definimos los **números naturales** (o **numerales de Church**) como sigue:

$$\begin{aligned}
\overline{0} &= \lambda x.\lambda y.y \\
\overline{1} &= \lambda x.\lambda y.xy \\
\overline{2} &= \lambda x.\lambda y.x(xy) \\
\overline{3} &= \lambda x.\lambda y.x(x(xy)) \\
&\vdots
\end{aligned}$$

En general, dado $m \in \mathbb{N}$, definimos

$$\overline{m} = \lambda x.\lambda y.\underbrace{x(x(\cdots x(x}_{m \text{ veces}} y)\cdots))$$

Denotemos por $\overline{\mathbb{N}}$ al conjunto de los numerales de Church. Claramente $\overline{\mathbb{N}} \subset \Lambda$.

Para simplificar la notación, establezcamos una notación: dada una variable x, definamos el contexto $x^1[*] = x(*)$ e inductivamente para $m \in \mathbb{N}$, $x^{m+1}[*] = x(x^m[*])$. Así, por ejemplo tenemos

$$x^1[y] = xy, \qquad x^2[y] = x(x^1[y]) = x(xy) \qquad x^3[y] = x(x^2[y]) = x(x(xy)).$$

Luego, en general, $\overline{m} = \lambda x.\lambda y.x^m[y]$.

Ahora definiremos las funciones aritméticas básicas. Por ejemplo, la función sucesor succ está definida como sigue:

$$\mathtt{succ} = \lambda x.\lambda y.\lambda z.y(xyz).$$

En efecto, dado $\overline{m} \in \overline{\mathbb{N}}$, tenemos

$$\begin{aligned}
\mathtt{succ}\ \overline{m} &= (\lambda x.\lambda y.\lambda z.y(xyz))\overline{m} \\
&\to_\beta \lambda y.\lambda z.y(\overline{m}yz) \\
&= \lambda y.\lambda z.y((\lambda w.\lambda r.w^m[r])yz) \\
&\to_\beta \lambda y.\lambda z.y((\lambda r.y^m[r])z) \\
&\to_\beta \lambda y.\lambda z.y(y^m[z]) \\
&=_\alpha \lambda x.\lambda y.x^{m+1}[y] \\
&= \overline{m+1}.
\end{aligned}$$

La función suma de naturales esta dada por el λ-término

$$\mathtt{add} = \lambda x.\lambda y.\lambda a.\lambda b.(xa)(yab).$$

En efecto, dados $\overline{m}, \overline{n}$ tenemos

$$\begin{aligned}
\texttt{add}\ \overline{m}\ \overline{n} &= (\lambda x.\lambda y.\lambda a.\lambda b.(xa)(yab))\ \overline{m}\ \overline{n} \\
&\to_\beta (\lambda y.\lambda a.\lambda b.(\overline{m}a)(yab))\ \overline{n} \\
&\to_\beta \lambda a.\lambda b.(\overline{m}a)(\overline{n}ab) \\
&= \lambda a.\lambda b.((\lambda x.\lambda y.x^m[y])a)((\lambda x.\lambda y.x^n[y])ab) \\
&\to_\beta \lambda a.\lambda b.(\lambda y.a^m[y])((\lambda y.a^n[y])b) \\
&\to_\beta \lambda a.\lambda b.(\lambda y.a^m[y])(a^n[b]) \\
&\to_\beta \lambda a.\lambda b.a^m[a^n[b]] \\
&= \lambda a.\lambda b.a^{m+n}[b] \\
&= \overline{m+n}.
\end{aligned}$$

Demostraciones análogas se tienen para probar que los λ-términos

$$\begin{aligned}
\texttt{pred} &= \lambda x.\lambda y.\lambda z.x(\lambda w.\lambda r.r(wy))(\lambda u.z)(\lambda v.v) \\
\texttt{rest} &= \lambda x.\lambda y.y\,\texttt{pred}\ x, \\
\texttt{mult} &= \lambda x.\lambda y.\lambda z.x(yz)
\end{aligned}$$

representan las operaciones de predecesor, resta y multiplicación, respectivamente.

Así, el λ-cálculo sirve para definir funciones naturales. Esto motiva la siguiente definición.

Definición 2.24. Diremos que una función $f : \mathbb{N}^r \to \mathbb{N}$ es **λ-calculable** si existe un λ-término M tal que

$$M\overline{m_1}\cdots\overline{m_r} \to_\beta^* \overline{f(m_1,\ldots,m_r)},$$

para todo $(m_1,\ldots,m_r) \in \mathbb{N}^r$.

El λ-cálculo también puede usarse para representar funciones booleanas y cálculo proposicional.

Definición 2.25. Definimos las **variables booleanas** $\mathcal{V}$ (representando verdadero) y $\mathcal{F}$ (representando falso) como

$$\begin{aligned}
\mathcal{V} &= \lambda x.\lambda y.x \\
\mathcal{F} &= \lambda x.\lambda y.y
\end{aligned}$$

Usando estos términos, podemos definir las funciones booleanas $\neg$, $\vee$, $\wedge$, respectivamente, como

$$\begin{aligned}\neg &= \lambda xyz.xzy\\ \vee &= \lambda xy.xxy\\ \wedge &= \lambda xy.xyx.\end{aligned}$$

Por ejemplo, considerando el λ-término $\neg\mathcal{V}$, tenemos

$$\begin{aligned}\neg\mathcal{V} &= (\lambda xyz.xzy)(\lambda r.\lambda s.r)\\ &\to_\beta \lambda y.\lambda z.(\lambda r.\lambda s.r)zy\\ &\to_\beta \lambda y.\lambda z.(\lambda s.z)y\\ &\to_\beta \lambda y.\lambda z.z\\ &= \mathcal{F}.\end{aligned}$$

Procediendo de la misma forma, podemos definir un λ-término que se comporta como un condicional. Definimos el λ-término `if` como

$$\texttt{if} = \lambda xyz.xyz.$$

Veamos su comportamiento. Consideremos dos λ-términos M y N, entonces

$$\begin{aligned}\texttt{if}\mathcal{V}MN &= (\lambda xyz.xyz)\,\mathcal{V}\,M\,N\\ &\to_\beta (\lambda yz.\mathcal{V}yz)M\,N\\ &= (\lambda yz.(\lambda rs.r)yz)M\,N\\ &\to_\beta (\lambda yz.(\lambda s.y)z)M\,N\\ &\to_\beta (\lambda yz.y)M\,N\\ &\to_\beta (\lambda z.M)N\\ &\to_\beta M.\end{aligned}$$

Análogamente, $\texttt{if}\mathcal{F}MN \to_\beta N$.

Usando el λ-término `if`, podemos definir un λ-término que verifica si un numero es o no cero:

$$\texttt{escero} = \lambda x.x(\lambda y.\mathcal{F})\mathcal{V}.$$

En efecto, para cualquier $\overline{m} \in \overline{\mathbb{N}}$,

$$\begin{aligned}
\texttt{escero}\overline{m} &= \lambda x.x(\lambda y.\mathcal{F})\mathcal{V}\overline{m} \\
&\rightarrow_\beta \overline{m}(\lambda y.\mathcal{F})\mathcal{V} \\
&\rightarrow_\beta (\lambda r.\lambda s.r^m[s])(\lambda y.\mathcal{F})\mathcal{V} \\
&\rightarrow_\beta (\lambda s.(\lambda y.\mathcal{F})^m[s])\mathcal{V} \\
&\rightarrow_\beta (\lambda y.\mathcal{F})^m[\mathcal{V}].
\end{aligned}$$

Donde la última expresión es igual a $\mathcal{V}$, si $\overline{m} = \overline{0}$, y se β-reduce a $\mathcal{F}$, en caso contrario.

2.2.4. Recursividad y puntos fijos en λ-términos

Usando los λ-términos definidos anteriormente, intentemos definir la función *factorial*.

$$\texttt{factorial} = \lambda n.\texttt{if}(\texttt{escero}\, n)\ \overline{1}\ \texttt{mult}\ (n\texttt{factorial}(\texttt{pred}\, n)).$$

Este λ-término no está bien definido, pues en su definición contiene una *llamada* a si mismo. Este problema puede ser evitado con el uso de *combinadores de punto fijo*. Un **combinador** es un λ-término Y tal que, para cualquier λ-término M,

$$YM = M(YM).$$

Un ejemplo de combinador fue dado por Curry [Church, 1932, Curry, 1934]. Este definió

$$Y = \lambda x.(\lambda y.x(yy))(\lambda y.x(yy)).$$

En efecto,

$$\begin{aligned}
YM &= (\lambda x.(\lambda y.x(yy))(\lambda y.x(yy)))M \\
&\rightarrow_\beta (\lambda y.M(yy))(\lambda y.M(yy)) \\
&\rightarrow_\beta M((\lambda y.M(yy))(\lambda y.M(yy))) \\
&= M(YM).
\end{aligned}$$

Usando este λ-término, podemos definir el *factorial* de la siguiente manera:

$$\texttt{factorial} = YF,$$

donde $F = \lambda f.\lambda n.\texttt{if}(\texttt{escero}\, n)\ \overline{1}\ \texttt{mult}\ (n f(\texttt{pred}\, n))$. En efecto,

$$\begin{aligned}
\texttt{factorial}\overline{n} &= (YF)\overline{n} \\
&\to_\beta (F(YF))\overline{n} \\
&\to_\beta ((\lambda f.\lambda n.\texttt{if}(\texttt{escero}\, n)\ \overline{1}\ \texttt{mult}\ (n f(\texttt{pred}\, n)))(YF))\overline{n} \qquad (2.1)\\
&\to_\beta (\lambda n.\texttt{if}(\texttt{escero}\, n)\ \overline{1}\ \texttt{mult}\ (n(YF)(\texttt{pred}\, n)))\overline{n} \\
&\to_\beta \texttt{if}(\texttt{escero}\, \overline{n})\ \overline{1}\ \texttt{mult}\ (\overline{n}(YF)(\texttt{pred}\, \overline{n}))
\end{aligned}$$

Probemos por inducción sobre $\overline{n}$ que el λ-término anterior se β-reduce a $\overline{n!}$. Si $\overline{n} = \overline{0}$, por definición de if, tenemos que $\texttt{factorial}\overline{n} = \overline{1}$. Supongamos que tenemos $\texttt{factorial}\overline{n} \to_\beta \overline{n!}$. Entonces $(\texttt{pred}\, \overline{n+1}) \to_\beta \overline{n}$ y, por (2.1),

$$\begin{aligned}
\texttt{factorial}\overline{n+1} &\to_{beta} \texttt{if}(\texttt{escero}\, \overline{n+1})\ \overline{1}\ \texttt{mult}\ (\overline{n+1}(YF)(\texttt{pred}\, \overline{n+1})) \\
&\to_{beta} \texttt{mult}\ (\overline{n+1}(YF)(\texttt{pred}\, \overline{n+1})) \\
&\to_{beta} \texttt{mult}\ (\overline{n+1}(YF)\overline{n}) \\
&\to_{beta} \texttt{mult}\ (\overline{n+1}\ \texttt{factorial}\overline{n}) \\
&\to_{beta} \texttt{mult}\ (\overline{n+1}\ \overline{n!}) \\
&\to_{beta} \overline{(n+1)!}.
\end{aligned}$$

2.3. Funciones recursivas

La teoría de funciones recursivas es una alternativa al λ-cálculo y fue dada por Gödel[1] y estudiada por Kleene en la década de 1930. Se basa en definir la familia de las funciones que se pueden obtener a partir de ciertas funciones *iniciales* y operaciones sobre estas.

2.3.1. Funciones recursivas primitivas

Definición 2.26. Definimos inductivamente las funciones **recursivas primitivas** como sigue:

1. Las siguientes funciones, que llamaremos **funciones iniciales**, son recursivas primitivas:

   a) La función cero, $0 : \mathbb{N} \to \mathbb{N}$, $0(n) = 0$.

   b) La función sucesor, $s : \mathbb{N} \to \mathbb{N}$, $s(n) = n + 1$.

   c) Las funciones proyección $\Pi^r_j : \mathbb{N}^r \to \mathbb{N}$, definida como

   $$\Pi^r_j(m_1, \ldots, m_r) = m_j,$$

   para cada $r \geqslant 1$ e $j = 1, \ldots, r$.

[1] En un curso dado por este en 1934 en la Universidad de Princeton.

2. Si $g : \mathbb{N}^r \to \mathbb{N}$ y $h_j : \mathbb{N}^s \to \mathbb{N}$, $j = 1, \ldots, r$, son recursivas primitivas, entonces $f : \mathbb{N}^s \to \mathbb{N}$ definida como
$$f(\overline{m}) = g(h_1(\overline{m}), \ldots, h_r(\overline{m})),$$
donde $\overline{m} \in \mathbb{N}^s$, es recursiva primitiva.

3. Si $g : \mathbb{N}^r \to \mathbb{N}$ y $h : \mathbb{N}^{r+2} \to \mathbb{N}$ son recursivas primitivas entonces $f : \mathbb{N}^{r+1} \to \mathbb{N}$ dada por
$$f(\overline{m}, 0) = g(\overline{m}),$$
$$f(\overline{m}, n+1) = h(\overline{m}, n, f(\overline{m}, n)),$$
donde $\overline{m} \in \mathbb{N}^r$, es recursiva primitiva.

Llamaremos por $\mathcal{PR}$ al conjunto de las funciones recursivas primitivas.

Las operaciones dadas en los ítems 2 y 3 de la definición anterior sirven para generar nuevas funciones recursivas primitivas a partir de funciones recursivas primitivas ya dadas. Estas operaciones se conocen como **substitución** y **recursión primitiva**, respectivamente. Así, $\mathcal{PR}$ se caracteriza por ser el menor conjunto de funciones que contiene a las funciones iniciales y ser cerrado bajo substitución y recursión primitiva. Además, es claro de la definición de estas operaciones y del hecho que las funciones iniciales son totalmente definidas, que toda función recursiva primitiva es totalmente definida.

Ejemplo 2.27. Las siguientes funciones son recursivas primitivas:

1. Para cada $k \in \mathbb{N}$, la función constante $f_k : \mathbb{N} \to \mathbb{N}$, $f_k(n) = k$.

2. La función suma, $+ : \mathbb{N}^2 \to \mathbb{N}$, $+(m, n) = m + n$.

3. La función producto, $\times : \mathbb{N}^2 \to \mathbb{N}$, $\times(m, n) = m \times n$.

4. La función potencia, $\wedge : \mathbb{N}^2 \to \mathbb{N}$, $\wedge(m, n) = m^n$.

5. La función predecesor, $\alpha : \mathbb{N} \to \mathbb{N}$,
$$\alpha(n) = \begin{cases} n-1, & \text{si } n > 0, \\ 0, & \text{si } n = 0. \end{cases}$$

6. La función diferencia restringida, $\dot{-} : \mathbb{N}^2 \to \mathbb{N}$,
$$\dot{-}(m, n) = m \dot{-} n = \begin{cases} m-n, & \text{si } m \geq n, \\ 0, & \text{si } m < n. \end{cases}$$

7. La función diferencia absoluta, $\boxminus : \mathbb{N}^2 \to \mathbb{N}$,

$$\boxminus = |m - n| = \begin{cases} m - n, & \text{si } m \geqslant n, \\ n - m, & \text{si } m < n. \end{cases}$$

8. Las funciones signo y signo inverso $\mathrm{sg}, \mathrm{sg}_- : \mathbb{N} \to \mathbb{N}$,

$$\mathrm{sg}(n) = \begin{cases} 0, & \text{si } n = 0, \\ 1, & \text{si } n \neq 0, \end{cases}$$

y $\mathrm{sg}_-(n) = 1 - \mathrm{sg}(n)$.

9. La función resto entero, $\mathrm{mod} : \mathbb{N}^2 \to \mathbb{N}$,

$$\mathrm{mod}(m, n) = \begin{cases} r \text{ tal que } m = qn + r,\ 0 \leqslant r < n, & \text{si } n \neq 0, \\ m, & \text{si } n = 0. \end{cases}$$

10. La función número de divisores, $D : \mathbb{N} \to \mathbb{N}$,

$$D(n) = \#\{d \,:\, 1 \leqslant d \leqslant n \text{ y } \mathrm{mod}(n, d) = 0\}.$$

2.3.2. La función de Ackermann

Mostramos a continuación una función no recursiva primitiva. Tal función se denomina función de Ackermann.

Definición 2.28. Definimos la **función de Ackermann**, $A : \mathbb{N}^2 \to \mathbb{N}$, como

$$\begin{aligned} A(m, 0) &= m + 1, \\ A(0, n + 1) &= A(1, n), \\ A(m + 1, n + 1) &= A(A(m, n + 1), n). \end{aligned}$$

Veamos algunas propiedades de esta función.

Proposición 2.29.

1. *A está bien definida en todo $\mathbb{N}^2$.*
2. $A(m, 1) = m + 2$.
3. $A(m, 2) = 2m + 3$.

4. $A(m,n) > m$.

5. $A(m,n) < A(m+1,n)$.

6. $A(m+1,n) \leqslant A(m,n+1)$.

7. $A(m,n) < A(m,n+1)$.

8. $A(A(m,n),p) < A(m,n+p+2)$.

Demostración:

1. Usaremos inducción sobre n. Para $n = 0$, tenemos que para todo $m \in \mathbb{N}$, $A(m,0) = m+1$. Luego $(m,0) \in \mathrm{dom}(A)$, para todo $m \in \mathbb{N}$. Ahora supongamos que dado $n \in \mathbb{N}$, $(m,n) \in \mathrm{dom}(A)$, para todo $m \in \mathbb{N}$. Probaremos, usando ahora inducción sobre m, que $(m,n+1) \in \mathrm{dom}(A)$, para todo $m \in \mathbb{N}$. En efecto, para $m = 0$, $A(0,n+1) = A(1,n)$, que implica que $(0,n+1) \in \mathrm{dom}(A)$. Supongamos ahora que $(m,n+1) \in \mathrm{dom}(A)$. Por la hipótesis inductiva original

$$(A(m,n+1),n) \in \mathrm{dom}(A)$$

luego $A(m+1,n+1) = A(A(m,n+1),n)$ está bien definida. Esto prueba que $\mathrm{dom}(A) = \mathbb{N}^2$.

2. Usaremos inducción sobre m. Para $m = 0$, por definición de A,

$$A(0,1) = A(1,0) = 2 = 0+2.$$

Supongamos ahora que $A(m,1) = m+2$, para cierto $m \in \mathbb{N}$. Entonces

$$A(m+1,1) = A(m+1,0+1) = A(A(m,1),0) = A(m,1)+1 = m+2+1 = (m+1)+2,$$

que es lo que queríamos demostrar.

3. Usando inducción sobre m. Para $m = 0$, por definición de A,

$$A(0,2) = A(1,1) = A(A(0,1),0) = A(0,1)+1 = 2+1 = 3.$$

Supongamos ahora que $A(m,2) = 2m+3$, para cierto $m \in \mathbb{N}$. Entonces

$$A(m+1,2) = A(A(m,2),1) = A(m,2)+2 = 2m+3+2 = 2(m+1)+3.$$

4. Usaremos inducción sobre n. Para $n = 0$, tenemos $A(m,0) = m+1 > m$, para todo $m \in \mathbb{N}$. Supongamos ahora que $A(m,n) > m$, para todo $m \in \mathbb{N}$. Ahora, por inducción sobre m, para $m = 0$,

$$A(0,n+1) = A(1,n) > 1 > 0.$$

Y en general, suponiendo $A(m, n+1) > m$ para cierto $m \in \mathbb{N}$, tenemos

$$A(m+1, n+1) = A(m+1, n+1) = A(A(m, n+1), n) > A(m, n+1) > m.$$

Desde que $A(m, n+1) \in \mathbb{N}$, entonces $A(m, n+1) \geqslant m+1$ y, por lo tanto,

$$A(m+1, n+1) > m+1.$$

5. Usaremos inducción sobre n. Para $n = 0$, tenemos que para todo $m \in \mathbb{N}$,

$$A(m, 0) = m+1 < m+2 = A(m+1, 0).$$

Supongamos que para cierto $n \in \mathbb{N}$, $A(m, n) < A(m+1, n)$, para todo $m \in \mathbb{N}$. Por el ítem 4, $A(m, n+1) < A(A(m, n+1), n)$. Luego, para todo $m \in \mathbb{N}$,

$$A(m, n+1) < A(A(m, n+1), n) = A(m+1, n+1).$$

6. Usaremos inducción sobre m. Para $m = 0$, la desigualdad se cumple trivialmente por la definición de A. Supongamos ahora que para cierto $m \in \mathbb{N}$, $A(m+1, n) \leqslant A(m, n+1)$, para todo $n \in \mathbb{N}$. Por el ítem 4, $m+1 < A(m+1, n)$, que equivale a $m+2 \leqslant A(m+1, n)$. Luego, por el ítem 4,

$$A(m+2, n) \leqslant A(A(m+1, n), n) \leqslant A(A(m, n+1), n) = A(m+1, n+1).$$

7. Usaremos inducción sobre n. Para $n = 0$, tenemos que para todo $m \in \mathbb{N}$,

$$A(m, 0) = m+1 < m+2 = A(m, 1).$$

Supongamos que para cierto $n \in \mathbb{N}$, $A(m, n) < A(m, n+1)$, para todo $m \in \mathbb{N}$. Si $m = 0$, tenemos

$$A(0, n+1) = A(1, n) < A(1, n+1) = A(0, n+2).$$

Si $m > 1$ entonces $m = p+1$ y así, usando los ítems 5 y 6,

$$\begin{aligned} A(m, n+1) = A(p+1, n+1) &= A(A(p, n+1), n) \\ &\leqslant A(A(p+1, n), n) \\ &= A(A(m, n), n) \\ &< A(A(m, n+1), n) \\ &= A(m+1, n+1). \end{aligned}$$

8. Fijados $m, n, p \in \mathbb{N}$,

$$\begin{aligned} A(A(m,n),p) &< A(A(m,n),p+n) \\ &< A(A(m,n+p+1),n+p) \\ &= A(m+1,n+p+1) \\ &\leqslant A(m,n+p+2). \end{aligned}$$

Mostraremos ahora que A no es recursiva primitiva. Para esto necesitamos la siguiente definición.

Definición 2.30. Una función $h : \mathbb{N}^2 \to \mathbb{N}$ se dice que **mayoriza** una función $g : \mathbb{N}^r \to \mathbb{N}$, si existe $b \in \mathbb{N}$ tal que

$$g(a_1, \ldots, a_r) < h(a, b),$$

para cualquier $(a_1, \ldots, a_r) \in \mathbb{N}^r$ con $a = \text{máx}\{a_1, \ldots, a_r\} > 1$.

Observación 2.31. Para el caso de la función de Ackermann, la condición $\text{máx}\{a_1, \ldots, a_r\} > 1$ dada en la definición 2.30 es superflua. Como $\{(a_1, \ldots, a_r) : \text{máx}\{a_1, \ldots, a_r\} \leqslant 1\}$ es finito y para cualquier $x, y \in \mathbb{N}$ la sucesión $(A(x, y+k))_k$ es creciente, entonces A mayoriza una función $g : \mathbb{N}^r \to \mathbb{N}$ si, y solo si, existe $b \in \mathbb{N}$ tal que

$$g(a_1, \ldots, a_r) < A(a, b),$$

para cualquier $(a_1, \ldots, a_r) \in \mathbb{N}^r$.

Consideremos el conjunto $\mathcal{A}$ de las funciones que son mayoradas por A. Probaremos tres lemas.

Lema 2.32. *Las funciones iniciales pertenecen a $\mathcal{A}$.*

Demostración:

1. Para la función cero, bastará tomar $b = 0$. Así, para todo a,

$$0(a) = 0 < A(a, 0) = a + 1.$$

   Luego $0 \in \mathcal{A}$.

2. Para la función sucesor, definamos $b = 1$. Entonces, para todo a,

$$s(a) = a + 1 < A(a, 1) = a + 2.$$

   Así, $s \in \mathcal{A}$.

3. Veamos el caso de las funciones proyección. Dados $r, j \in \mathbb{N}, r \geqslant 1$ y $1 \leqslant j \leqslant r$, sean $a_1, \ldots, a_r \in \mathbb{N}$. Entonces, eligiendo $b = 0$,

$$\Pi_j^r(a_1, \ldots, a_r) = a_j \leqslant a < a + 1 = A(a, 0),$$

que prueba que $\Pi_j^r \in \mathcal{A}$. □

Lema 2.33. *Si $g : \mathbb{N}^r \to \mathbb{N}$ y $h_j : \mathbb{N}^s \to \mathbb{N}$, $j = 1, \ldots, r$, pertenecen a $\mathcal{A}$, entonces $f : \mathbb{N}^s \to \mathbb{N}$ definida como*

$$f(\overline{m}) = g(h_1(\overline{m}), \ldots, h_r(\overline{m})),$$

donde $\overline{m} \in \mathbb{N}^s$, también pertenece a $\mathcal{A}$.

Demostración: Sean $a_1, \ldots, a_s \in \mathbb{N}$. Como cada $h_j \in \mathcal{A}$ entonces, para cada $j = 1, \ldots, r$, existe b_j tal que

$$h_j(a_1, \ldots, a_s) < A(a, b_j).$$

Por otro lado, como $g \in \mathcal{A}$, existe $b \in \mathbb{N}$ tal que, para todo $(x_1, \ldots, x_r) \in \mathbb{N}^r$,

$$g(x_1, \ldots, x_r) < A(x, b),$$

donde $x = \text{máx}\{x_1, \ldots, x_r\}$.

Sea i tal que

$$h_i(a_1, \ldots, a_s) = \text{máx}\{h_1(a_1, \ldots, a_s), \ldots, h_r(a_1, \ldots, a_s)\}.$$

Entonces

$$\begin{aligned} f(a_1, \ldots, a_s) &= g(h_1(a_1, \ldots, a_s), \ldots, h_r(a_1, \ldots, a_s)) \\ &< A(h_i(a_1, \ldots, a_s), b) \\ &< A(A(a, b_i), b) \\ &< A(a, b_i + b + 2), \end{aligned}$$

donde la ultima inecuación se debe al ítem 8 de la proposición 2.29. Esto implica que $f \in \mathcal{A}$. □

Lema 2.34. *Si $g : \mathbb{N}^r \to \mathbb{N}$ y $h : \mathbb{N}^{r+2} \to \mathbb{N}$ pertenecen a $\mathcal{A}$ entonces $f : \mathbb{N}^{r+1} \to \mathbb{N}$ dada por*

$$\begin{aligned} f(\overline{m}, 0) &= g(\overline{m}), \\ f(\overline{m}, n + 1) &= h(\overline{m}, n, f(\overline{m}, n)), \end{aligned}$$

donde $\overline{m} \in \mathbb{N}^r$, también pertenece a $\mathcal{A}$.

Demostración: Sean b_g y b_h las constantes dadas por la definición 2.30 (teniendo en cuenta la observación 2.31), para g y h, respectivamente. Afirmamos los siguiente: existe $q \in \mathbb{N}$ tal que, para todo $m_1, \ldots, m_r \in \mathbb{N}$ y $n \in \mathbb{N}$,

$$f(m_1, \ldots, m_r, n) < A(n + m, q),$$

donde $m = \text{máx}\{m_1, \ldots, m_r\}$. En efecto, definamos $q = 1 + \text{máx}\{b_g, b_h\} \geqslant 1$ y usemos inducción sobre n. Para $n = 0$, tenemos

$$f(m_1, \ldots, m_r, 0) = g(m_1, \ldots, m_r) < A(m, b_g) < A(m, q) = A(0 + m, q).$$

En general, supongamos que para cierto $n \in \mathbb{N}$, $f(m_1, \ldots, m_r, n) < A(n+m, q)$, para todo $(m_1, \ldots, m_r) \in \mathbb{N}^r$. Entonces

$$f(m_1, \ldots, m_r, n + 1) = h(m_1, \ldots, m_r, n, f(m_1, \ldots, m_r, n)) < A(z, b_h),$$

donde $z = \text{máx}\{m, n, f(m_1, \ldots, m_r, n)\}$. Como $\text{máx}\{m, n\} < m+n < A(m+n, q)$ y $f(m_1, \ldots, m_r, n) < A(m + n, q)$, por hipótesis inductiva, tenemos $z < A(m + n, q)$ y, por lo tanto,

$$f(m_1, \ldots, m_r, n + 1) < A(z, b_h) < A(A(m + n, q), q - 1) = A(m + n + 1, q).$$

Esto prueba la afirmación.

Finalmente, sean $a_1, \ldots, a_r, a_{r+1}$ y $a = \text{máx}\{a_1, \ldots, a_{r+1}\}$. Entonces, definiendo $a' = \text{máx}\{a_1, \ldots, a_r\}$,

$$\begin{aligned} f(a_1, \ldots, a_r, a_{r+1}) &< A(a_{r+1} + a', q) \\ &\leqslant A(2a, q) \\ &< A(2a + 3, q) \\ &= A(A(a, 2), q) \\ &< A(a, q + 4), \end{aligned}$$

donde hemos usado los ítems 3 y 8 de la proposición 2.29, en la cuarta y quinta desigualdad. □

Como $\mathcal{PR}$ es el menor conjunto de funciones que contiene a las funciones iniciales y es cerrado bajo substitución y recursión primitiva, entonces $\mathcal{PR} \subset \mathcal{A}$.

Proposición 2.35. *El conjunto de funciones recursivas primitivas, $\mathcal{PR}$, está contenido en $\mathcal{A}$. Concluimos que $A \notin \mathcal{PR}$.*

2.3.3. El operador μ, recursividad parcial y recursividad

Definición 2.36. Sea $g : \mathrm{dom}(g) \subset \mathbb{N}^{r+1} \to \mathbb{N}$ una función y $\overline{m} \in \mathbb{N}^r$. Definimos el **operador** μ u **operador de minimización** como

$$\mu[g(\overline{m}, \cdot) = 0] = \mathrm{mín}\{t \in \mathbb{N} : (\overline{m}, t) \in \mathrm{dom}(g) \text{ y } g(\overline{m}, t) = 0\}.$$

Vemos de la definición que el operador μ solo está definido para aquellos $\overline{m} \in \mathbb{N}^r$ tales que el conjunto $\{t \in \mathbb{N} : (\overline{m}, t) \in \mathrm{dom}(g) \text{ y } g(\overline{m}, t) = 0\}$ es no vacío. Luego, para $g : \mathrm{dom}(g) \subset \mathbb{N}^{r+1} \to \mathbb{N}$, podemos definir una función $f : \mathbb{N}^r \to \mathbb{N}$ definida como

$$f(\overline{m}) = \mu[g(\overline{m}, \cdot) = 0].$$

En este caso, $\overline{m} \in \mathrm{dom}(f)$ si, y solamente si, $\{t \in \mathbb{N} : (\overline{m}, t) \in \mathrm{dom}(g) \text{ y } g(\overline{m}, t) = 0\} \neq \varnothing$.

Estamos en condiciones de dar la siguiente definición.

Definición 2.37. Diremos que una función es **recursivas parcial** si:

1. es una de las funciones iniciales;
2. es generada por la operación de substitución aplicada en funciones recursivas parciales;
3. es generada por la operación de recursión primitiva aplicada en funciones recursivas parciales;
4. es generada por la operación μ aplicada en una función recursiva parcial.

Definición 2.38. Diremos que una función es **recursiva** si es recursiva parcial y es totalmente definida.

Ejemplo 2.39. La función de Ackermann, definida en la sección 2.3.2, es recursiva. La prueba de este hecho se escapa al objetivo de este trabajo, pero el lector interesado puede encontrarla en [Hermes, 1965, p. 90] y [Cutland, 1980, p. 194].

2.3.4. Recursividad de conjuntos

Sean $S \subset \mathbb{N}^r$. Definimos la **función característica**, $\chi_S : \mathbb{N}^r \to \mathbb{N}$, como

$$\chi_S(\overline{n}) = \begin{cases} 1, & \text{si } \overline{n} \in S, \\ 0, & \text{si } \overline{n} \notin S. \end{cases}$$

Definición 2.40. Diremos que $S \subset \mathbb{N}^r$ es **recursiva** (resp. **recursiva primitiva**) si χ_S lo es.

En particular, una relación $\mathcal{R}$ es una relación en $\mathbb{N}$ es recursiva (resp. primitiva recursiva) si

$$\xi_{\mathcal{R}}(m, n) = \begin{cases} 1, & \text{si } (m, n) \in \mathcal{R}, \\ 0, & \text{si } (m, n) \notin \mathcal{R} \end{cases}$$

lo es.

Ejemplo 2.41. La relación de igualdad $=$ en $\mathbb{N}$ es recursiva primitiva. Esto sigue de

$$\chi_{=}(m,n) = \mathrm{sg}_{-}(|m \dot{-} n|),$$

y del hecho que sg_{-} y $\dot{-}$ son recursivas primitivas.

Ejemplo 2.42. El conjunto de los números primos P en $\mathbb{N}$ es recursivo primitivo. Esto sigue de

$$\chi_P(n) = \mathrm{sg}_{-}(D(n) \dot{-} 2) \times \mathrm{sg}(n \dot{-} 1)$$

y del hecho que las funciones involucradas son recursivas primitivas.

2.4. Máquinas de Turing

2.4.1. Definición y ejemplos

Comenzaremos dando una descripción informal de máquina de Turing. Una máquina de Turing consiste de un *cabezal* de lectura/escritura, esto es, un autómata que actúa sobre un arreglo unidimensional, infinito en ambas direcciones, al cual llamaremos *cinta*. Las casillas del arreglo pueden estar vacías o pueden contener elementos de cierto conjunto que será llamado *alfabeto*. El cabezal de la máquina tiene asociado un *estado* y actúa bajo un conjunto de instrucciones previamente establecidas.

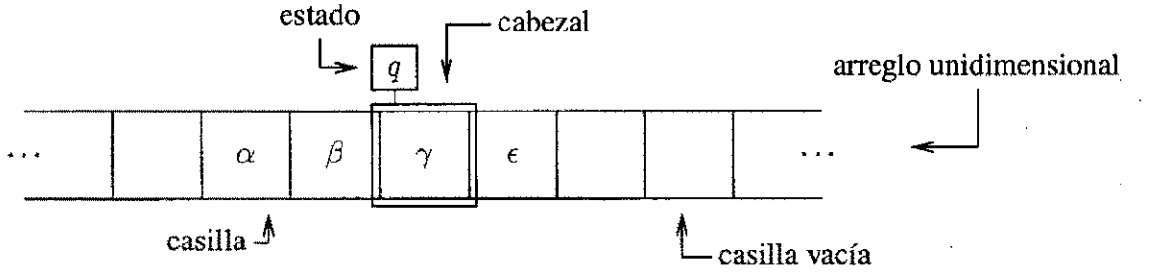


Figura 2.1: Representación de una máquina de Turing

Bajo estas hipótesis, un *paso* (o *acción*) de la máquina de Turing es el conjunto de las siguientes acciones:

1. el cabezal lee el contenido de la casilla a la que apunta,
2. el cabezal escribe o modifica, quizás, el contenido de dicha casilla,
3. el cabezal cambia, quizás, su estado,
4. el cabezal se mueve a la casilla de la derecha o de la izquierda.

Inicialmente, todas las casillas de la cinta de la máquina de Turing estarán vacías (también diremos que estarán "en blanco") excepto por una cantidad finita de ellas. El cabezal estará en un estado llamado "estado inicial" y apuntará a la casilla no vacía que se encuentra más a la izquierda. El contenido de las casillas no vacías antes de la ejecución se denominará "entrada" de la máquina de Turing.

Una "ejecución" de la máquina de Turing consistirá en dejar que el autómata *actúe* (es decir, ejecute un paso o acción) sobre el arreglo, tantas veces como sea necesario, hasta detenerse (pudiendo incluso nunca llegar a detenerse).

La ejecución de la máquina de Turing se detendrá cuando el cabezal adquiera un estado "final". En este caso, el estado del cabezal y el contenido de las casillas no vacías al final de la ejecución, serán la "salida" de la máquina de Turing. Como en cada paso el cabezal de la máquina modifica el contenido de sólo una casilla, entonces en cada paso de la ejecución de la máquina de Turing todas, salvo una cantidad finita de casillas, estarán vacías.

Las acciones de una máquina de Turing dependen de la información que hay en la casilla que el cabezal lee y del estado en que este se encuentra. Esto puede ser escrito de la siguiente manera

$$\begin{pmatrix} \text{información actual} \\ \text{estado actual} \end{pmatrix} \longmapsto \begin{pmatrix} \text{nueva información} \\ \text{nuevo estado} \\ \text{nueva posición relativa} \end{pmatrix}$$

Lo anterior da pie a la definición formal de máquina de Turing.

Definición 2.43. Una **máquina de Turing** M es una 7-upla $(\Sigma, \Gamma, Q, \delta, \flat, q_0, F)$ tal que

1. Γ es un conjunto finito y lo llamaremos **alfabeto** de M;
2. Q es un conjunto finito y lo llamaremos **conjunto de estados**. Este conjunto está conformado por los estados que el cabezal de la máquina puede adoptar;
3. Γ y Q son disjuntos;
4. $\flat$ es un elemento distinguido de $\Gamma \backslash \Sigma$. Este denotará el hecho que una casilla esté en blanco.
5. q_0 es un estado distinguido de Q al que llamaremos **estado inicial**. Este denotará el estado inicial de M antes de iniciar su ejecución.
6. $\Sigma \subset \Gamma \backslash \{\flat\}$ y lo llamaremos **alfabeto de entrada**. Antes de comenzar la ejecución de M, una cantidad finita de casillas no contienen $\flat$, sino elementos de Σ.
7. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$ es una función que denominaremos **función de transición**. Las instrucciones del cabezal están dadas por esta función, donde $\delta(q, r) = (q', r', \lambda)$ significa que el cabezal estando en estado q y encontrando a r en la casilla a donde apunta, reemplaza el contenido por r', cambia su estado a q' y se mueve a la izquierda si $\lambda = -1$ o a la derecha si $\lambda = 1$.

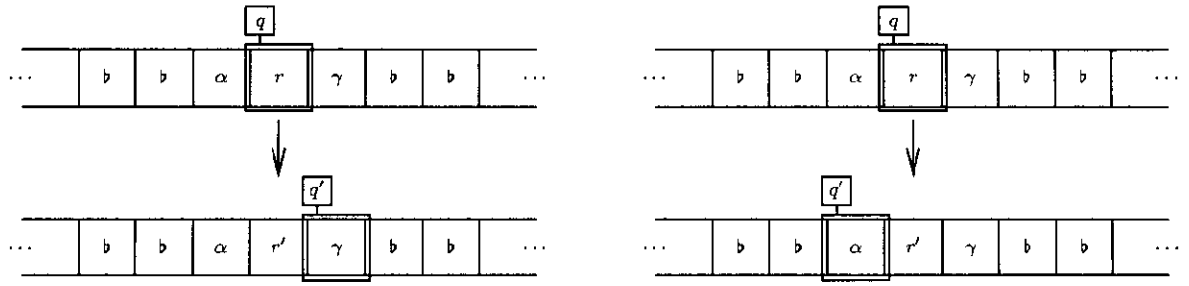


Figura 2.2: $\delta(q, r) = (q', r', 1)$ (derecha) y $\delta(q, r) = (q', r', -1)$ (izquierda)

8. $F \subset Q$ y lo llamaremos **conjunto de estados finales**. La ejecución de M se detendrá si en algún momento el cabezal cambia su estado a algún $q \in F$.

Como en cada momento de la ejecución de M sólo hay finitas casillas no vacías, entonces el contenido de la cinta de M puede ser representado por una cadena $x \in \Gamma^*$. Notemos que en la definición 2.43 no hemos mencionado un equivalente formal para el concepto de "cinta". Veremos que esto no es necesario; de hecho, la noción de cinta de una máquina de Turing es reemplazada por la siguiente definición.

Definición 2.44. Sea M una máquina de Turing. Una **descripción instantánea** de M es una cadena $C = xqy \in \Gamma^* Q \Gamma^*$, tal que $y \neq \epsilon$ y $q \in Q$.

Una descripción instantánea $C = xqy$ representa el hecho de tener la cadena xy escrita en la cinta de M con el cabezal en estado q apuntando a una casilla conteniendo el primer elemento de la cadena y. La siguiente definición modela un "paso" de M.

Definición 2.45. Dada C una descripción instantánea de M, denotaremos $C \xmapsto{M} C'$ si, escribiendo $C = xqsy$ con $x, y \in \Gamma^*$, $s \in \Gamma$ y $q \in Q$, y teniéndose $\delta(q, s) = (q', s', \lambda)$, entonces

- Si $\lambda = 1$ entonces $C' = xs'q'y$ en caso $y \neq \epsilon$, y $C' = xs'q'\flat$ en caso $y = \epsilon$.
- Si $\lambda = -1$ entonces $C' = x'q'as'y$ en caso $x = x'a$, y $C' = q'\flat s'y$ en caso $x = \epsilon$.

En este caso, C' también es una descripción instantánea de M.

Observe que C' está únicamente determinado por C y de M.

Definición 2.46. Diremos que una descripción instantánea C es **inicial** si $C = q_0 w$, $w \in \Sigma^* \backslash \{\varnothing\}$. Diremos que es **final** si $C = xqy$ con $q \in F$, $x, y \in \Gamma^*$.

Estamos listos para formalizar el concepto de "ejecución" de una máquina de Turing.

Definición 2.47. Una **ejecución** con entrada $w_0 \in \Sigma^*$ de una máquina de Turing M es una sucesión de descripciones instantáneas de M, $(C_k)_{k \in I}$, con $I = \{1, \ldots, n\}$ o $I = \mathbb{N}$, tales que

1. $C_0 = q_0 w_0$, es decir C_0 es inicial.

2. $C_k \overset{M}{\longmapsto} C_{k+1}$, para todo $k \in I$.

3. Si I es finito, entonces C_n es final.

En caso I sea finito, diremos que la ejecución de M se **detiene** con entrada w_0, y que la descripción instantánea final C_n es la **salida** de M.

Observación 2.48. Necesitamos que se cumplan tres propiedades adicionales en una máquina de Turing M:

1. El dominio de δ será suficientemente grande de tal manera que para toda entrada $w_0 \in \Sigma^*$, una ejecución de M con entrada w_0 esté bien definida (independientemente de si es finita o infinita).

2. Ejecuciones finitas de M poseen una única descripción instantánea final.

3. Ejecuciones infinitas de M no poseen descripciones instantáneas finales.

Estrictamente hablando, tales propiedades no se desprenden de las definiciones anteriores. Por ejemplo, en la definición de máquina de Turing, nada impide que el dominio de δ sea vacío, o que δ esté definida para algún par (q, r) con $q \in F$. Para obtener tales propiedades, deberíamos imponer condiciones adicionales en la definiciones 2.43 y 2.47, pero no lo haremos por cuestiones de compatibilidad con la literatura, y para mantener cierta simplicidad en tales definiciones. Sin embargo, por completitud, enunciaremos condiciones suficientes para obtener cada propiedad.

Proposición 2.49. *Sea M una máquina de Turing. Para que se cumpla la propiedad 1 de la observación 2.48 es suficiente que $(Q\backslash F) \times \Gamma \subset \mathrm{dom}(\delta)$. Además, para que se cumplan las propiedades 2 y 3 de la observación anterior, es suficiente tener que $\mathrm{dom}(\delta) \cap (F \times \Gamma) = \varnothing$. Finalmente, ambas condiciones se obtienen de la condición $\mathrm{dom}(\delta) = (Q\backslash F) \times \Gamma$.*

Demostración: El hecho que $(Q\backslash F) \times \Gamma \subset \mathrm{dom}(\delta)$ quiere decir que para todo estado no final y para todo caracter del alfabeto Γ, la función δ está bien definida. Esto quiere decir que para cualquier descripción instantánea C de M (en particular una descripción instantánea inicial), es posible construir una descripción instantánea C' tal que $C \overset{M}{\longmapsto} C'$. Luego, para todo $w_0 \in \Sigma^*$, una ejecución de M con entrada w_0 estará bien definida.

Por otro lado, $\mathrm{dom}(\delta) \cap (F \times \Gamma) = \varnothing$ significa que para todo estado final q, $\delta(q, r)$ no está definida para ningún caracter $r \in \Gamma$. Luego, si $(C_k)_{k \in I}$ es una ejecución de M y $C_{k'}$ es una descripción instantánea final de M entonces no puede darse $C_{k'} \overset{M}{\longmapsto} C_{k'+1}$ a menos que I sea finito y k' sea el último elemento de I. Esto implica las propiedades 2 y 3. □

Claramente, ambas condiciones dadas arriba no son necesarias. El ejemplo 4 más adelante muestra que la condición $(Q\backslash F) \times \Gamma \subset \mathrm{dom}(\delta)$ no lo es. Además, si en una máquina M que cumple las propiedades 2 y 3, agregamos estados finales "inútiles" a F que nunca son alcanzados y sin embargo δ

2. $C_k \overset{M}{\longmapsto} C_{k+1}$, para todo $k \in I$.

3. Si I es finito, entonces C_n es final.

En caso I sea finito, diremos que la ejecución de M se **detiene** con entrada w_0, y que la descripción instantánea final C_n es la **salida** de M.

Observación 2.48. Necesitamos que se cumplan tres propiedades adicionales en una máquina de Turing M:

1. El dominio de δ será suficientemente grande de tal manera que para toda entrada $w_0 \in \Sigma^*$, una ejecución de M con entrada w_0 esté bien definida (independientemente de si es finita o infinita).

2. Ejecuciones finitas de M poseen una única descripción instantánea final.

3. Ejecuciones infinitas de M no poseen descripciones instantáneas finales.

Estrictamente hablando, tales propiedades no se desprenden de las definiciones anteriores. Por ejemplo, en la definición de máquina de Turing, nada impide que el dominio de δ sea vacío, o que δ esté definida para algún par (q, r) con $q \in F$. Para obtener tales propiedades, deberíamos imponer condiciones adicionales en la definiciones 2.43 y 2.47, pero no lo haremos por cuestiones de compatibilidad con la literatura, y para mantener cierta simplicidad en tales definiciones. Sin embargo, por completitud, enunciaremos condiciones suficientes para obtener cada propiedad.

Proposición 2.49. *Sea M una máquina de Turing. Para que se cumpla la propiedad 1 de la observación 2.48 es suficiente que $(Q\backslash F) \times \Gamma \subset \mathrm{dom}(\delta)$. Además, para que se cumplan las propiedades 2 y 3 de la observación anterior, es suficiente tener que $\mathrm{dom}(\delta) \cap (F \times \Gamma) = \varnothing$. Finalmente, ambas condiciones se obtienen de la condición $\mathrm{dom}(\delta) = (Q\backslash F) \times \Gamma$.*

Demostración: El hecho que $(Q\backslash F) \times \Gamma \subset \mathrm{dom}(\delta)$ quiere decir que para todo estado no final y para todo caracter del alfabeto Γ, la función δ está bien definida. Esto quiere decir que para cualquier descripción instantánea C de M (en particular una descripción instantánea inicial), es posible construir una descripción instantánea C' tal que $C \overset{M}{\longmapsto} C'$. Luego, para todo $w_0 \in \Sigma^*$, una ejecución de M con entrada w_0 estará bien definida.

Por otro lado, $\mathrm{dom}(\delta) \cap (F \times \Gamma) = \varnothing$ significa que para todo estado final q, $\delta(q, r)$ no está definida para ningún caracter $r \in \Gamma$. Luego, si $(C_k)_{k \in I}$ es una ejecución de M y $C_{k'}$ es una descripción instantánea final de M entonces no puede darse $C_{k'} \overset{M}{\longmapsto} C_{k'+1}$ a menos que I sea finito y k' sea el último elemento de I. Esto implica las propiedades 2 y 3. □

Claramente, ambas condiciones dadas arriba no son necesarias. El ejemplo 4 más adelante muestra que la condición $(Q\backslash F) \times \Gamma \subset \mathrm{dom}(\delta)$ no lo es. Además, si en una máquina M que cumple las propiedades 2 y 3, agregamos estados finales "inútiles" a F que nunca son alcanzados y sin embargo δ

4. $Q = \{q_0, q_1, q_2, q_3, q_4\} \cup F$.

5. δ está definida de la siguiente manera

δ	0	|	χ	$\flat$
q_0	$(q_1, \chi, 1)$	$(q_4, \|, 1)$		
q_1	$(q_1, 0, 1)$	$(q_1, \|, 1)$	$(q_2, \chi, -1)$	$(q_2, \flat, -1)$
q_2	$(q_3, \chi, -1)$	$(q_n, \|, 1)$		
q_3	$(q_3, 0, -1)$	$(q_3, \|, -1)$	$(q_0, \chi, 1)$	
q_4	$(q_n, 0, 1)$		$(q_s, \chi, 1)$	$(q_s, \flat, -1)$

La máquina M permite entradas de la forma $0^n|0^m$ y devuelve q_s si $n = m$, y q_n en caso contrario. La máquina "marca" (estados q_0 y q_2) sucesivamente el primer y el último elemento no marcado de la entrada y detecta (estados q_2 y q_4) cuando ha habido una mayor cantidad de marcas de un lado que de otro, o cuando hay igual cantidad de marcas.

Por ejemplo, sea $w = 00|0$. Entonces lo siguiente es una ejecución de M:

$$\begin{aligned} q_000|0 &\overset{M}{\longmapsto} \chi q_1 0|0 \overset{M}{\longmapsto} \chi 0 q_1|0 \overset{M}{\longmapsto} \chi 0|q_1 0 \overset{M}{\longmapsto} \chi 0|0 q_1 \flat \\ &\overset{M}{\longmapsto} \chi 0|q_2 0 \overset{M}{\longmapsto} \chi 0 q_3|\chi \overset{M}{\longmapsto} \chi q_3 0|\chi \overset{M}{\longmapsto} q_3 \chi 0|\chi \\ &\overset{M}{\longmapsto} \chi q_0 0|\chi \overset{M}{\longmapsto} \chi\chi q_1|\chi \overset{M}{\longmapsto} \chi\chi|q_1\chi \overset{M}{\longmapsto} \chi\chi q_2|\chi \\ &\overset{M}{\longmapsto} \chi\chi q_2|\chi \overset{M}{\longmapsto} \chi\chi|q_n\chi, \end{aligned}$$

de donde, como mencionamos anteriormente, tenemos que $f_M(00|0) = (\chi\chi|\chi, q_n)$.

Ejemplo 2.52. Construiremos ahora una máquina de Turing M tal que dado $a \in \mathbb{N}$, M con entrada $a_{(2)}$ devuelva $c_{(2)}$, donde $c = a + 1$. Definamos la siguiente máquina de Turing: $M = (\Sigma, \Gamma, Q, \delta, \flat, q_0, F)$ tal que

1. $\Gamma = \{0, 1, \flat\}$.

2. $\Sigma = \{0, 1\}$.

3. $Q = \{q_0, q_1, q_f\}$.

4. $F = \{q_f\}$.

5. δ está definida de la siguiente manera

δ	0	1	$\flat$
q_0	$(q_0, 0, 1)$	$(q_0, 1, 1)$	$(q_1, \flat, -1)$
q_1	$(q_f, 1, -1)$	$(q_1, 0, -1)$	$(q_f, 1, 1)$

Dado un número $a_1 \cdots a_r$ escrito en binario, para construir la respuesta debemos reemplazar todos los 1 consecutivos encontrados desde la derecha por 0, hasta hallar un 0 o el símbolo $\flat$, los cuales reemplazaremos por 1 y detendremos la ejecución. De la definición de δ, vemos que el estado q_0 mueve el cabezal hacia el final de la entrada, sin modificarla. Luego, el estado q_1 reemplaza todos los 1 que encuentra por 0 moviéndose a la izquierda, hasta llegar o bien al primer 0 o a una casilla vacía, donde coloca un 1. Luego M calcula efectivamente $c = a + 1$.

Sea $w = 1011$, entonces lo siguiente es una ejecución de M:

$$\begin{aligned} q_0 1011 &\overset{M}{\longmapsto} 1 q_0 011 \overset{M}{\longmapsto} 10 q_0 11 \overset{M}{\longmapsto} 101 q_0 1 \\ &\overset{M}{\longmapsto} 1011 q_0 \flat \overset{M}{\longmapsto} 101 q_1 1 \overset{M}{\longmapsto} 10 q_1 10 \\ &\overset{M}{\longmapsto} 1 q_1 000 \overset{M}{\longmapsto} q_f 1100 \end{aligned}$$

de donde tenemos que $f_M(1011) = (1100, q_f)$.

Ejemplo 2.53. Sea $\Sigma = \{1, \ldots, n\}$. Construiremos una máquina de Turing tal que, dado $w \in \Sigma^*$, construya el sucesor de w según el orden lexicográfico de Σ^*. Definamos la siguiente máquina de Turing: $M = (\Sigma, \Gamma, Q, \delta, \flat, q_0, F)$ tal que

1. $\Sigma = \{1, \ldots, n\}$.
2. $\Gamma = \Sigma \cup \{\flat\}$.
3. $Q = \{q_0, q_1, q_f\}$.
4. $F = \{q_f\}$.
5. δ está definida de la siguiente manera

δ	$r \in \Sigma \backslash \{n\}$	n	$\flat$
q_0	$(q_0, r, 1)$	$(q_0, n, 1)$	$(q_1, \flat, -1)$,
q_1	$(q_f, r', 1)$	$(q_1, 1, -1)$	$(q_f, 1, 1)$

donde, dado $r \in \Sigma$, denotamos $r' = r + 1$.

Ejemplo 2.54. Ahora atacaremos el problema de construir una máquina M tal que dado $a \in \mathbb{N}$ entonces M, con entrada $a_{(2)}$, devuelva $c_{(2)}$, donde $c = a - 1$. Por simplicidad, admitiremos el caso que 0 sea un prefijo de la representación binaria de a (es decir, las cadenas de entrada y/o salida tengan uno o más ceros a la izquierda). Definamos la siguiente máquina de Turing: $M = (\Sigma, \Gamma, Q, \delta, \flat, q_0, F)$ tal que

1. $\Gamma = \{0, 1, \flat\}$.
2. $\Sigma = \{0, 1\}$.

3. $Q = \{q_0, q_1, q_f\}$.

4. $F = \{q_f\}$.

5. δ está definida de la siguiente manera

δ	0	1	b
q_0	$(q_0, 0, 1)$	$(q_0, 1, 1)$	$(q_1, \mathtt{b}, -1)$
q_1	$(q_1, 1, -1)$	$(q_f, 0, -1)$	$(q_f, \mathtt{b}, 1)$

Ejemplo 2.55. Finalmente, dados $a, b \in \mathbb{N}$, queremos encontrar una máquina M que con entrada $a_{(2)} + b_{(2)} \in \{0, 1, +\}^*$ devuelva $c_{(2)}$ tal que $c = a + b$. Definamos la siguiente máquina de Turing: $M = (\Sigma, \Gamma, Q, \delta, \mathtt{b}, q_0, F)$ tal que

1. $\Gamma = \{0, 1, +, \mathtt{b}\}$.

2. $\Sigma = \{0, 1, +\}$.

3. $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_f\}$.

4. $F = \{q_f\}$.

5. δ está definida de la siguiente manera

δ	0	1	+	b
q_0	$(q_0, 0, 1)$	$(q_0, 1, 1)$	$(q_1, +, -1)$	
q_1	$(q_2, 1, -1)$	$(q_1, 0, -1)$		$(q_2, 1, 1)$
q_2	$(q_2, 0, 1)$	$(q_2, 1, 1)$	$(q_2, +, 1)$	$(q_3, \mathtt{b}, -1)$
q_3	$(q_3, 1, -1)$	$(q_4, 0, -1)$	$(q_5, +, -1)$	
q_4	$(q_4, 0, -1)$	$(q_4, 1, -1)$	$(q_1, +, -1)$	
q_5	$(q_5, 0, 1)$	$(q_5, 1, 1)$		$(q_6, \mathtt{b}, -1)$
q_6	$(q_6, 0, -1)$	$(q_4, 1, -1)$	$(q_7, \mathtt{b}, 1)$	
q_7	$(q_7, \mathtt{b}, 1)$			$(q_f, \mathtt{b}, -1)$

La máquina M funciona de la siguiente manera: los estados q_0 y q_1 suman 1 al número a (vea el ejemplo del ítem 2.52. Luego los estados q_2 y q_3 se encargan de restar 1 al número b (vea el ejemplo del ítem 2.54. El estado q_4 se encarga de poner el cabezal en posición de tal manera de sumar 1 otra vez a a. Los estados q_5 y q_6 se encargan de determinar si b es aún mayor que cero, mientras que el estado q_7 limpia las casillas de la cinta a la derecha del símbolo + cuando ya se tiene que $b = 0$.

De los ejemplos anteriores podemos observar que para tareas relativamente simples (comparación de cadenas, suma de dos números), la descripción formal de una máquina de Turing que realiza tales

tareas puede ser bastante complicada. Esto podría poner en tela de juicio el "poder" de las máquinas de Turing, es decir, la cantidad de tareas que son realmente capaces de realizar. Recordemos que el motivo de definir máquinas de Turing fue el de formalizar el concepto de algoritmo; entonces debemos convencernos de que todo problema que puede ser resuelto usando algoritmos, también puede ser resuelto usando máquinas de Turing. Discutiremos más sobre este problema en la sección 2.5.

En adelante, para simplificar el tratamiento de las máquinas de Turing, utilizaremos una descripción de "alto nivel" al momento de definirlas, es decir, reemplazaremos la definición formal (7-uplas y funciones de transición) con descripciones breves de los pasos que ejecuta la máquina de Turing. En los ejemplos anteriores, usamos tanto como una descripción de alto nivel como la definición formal.

Ahora definiremos una generalización de la máquina de Turing.

2.4.2. Máquina de Turing de múltiples cintas

Una máquina de Turing de múltiples cintas es como una máquina de Turing ordinaria con varios cabezales que apuntan a varias cintas a la vez. Inicialmente, la entrada se encuentra en la primera cinta y las demás cintas se encuentran en blanco. La función de transición permite leer y escribir en cada cinta y mover cada cabezal al mismo tiempo, dependiendo sólo del estado de la máquina y lo leído por cada cabezal. Formalmente

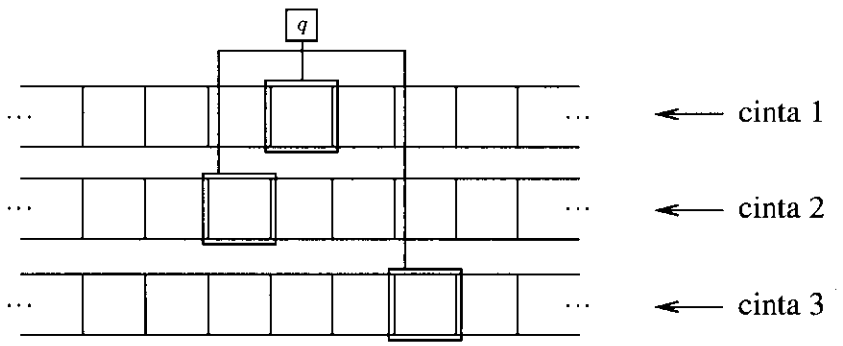


Figura 2.3: Representación de una máquina de Turing de múltiples cintas

Definición 2.56. Una **máquina de Turing de k cintas** es una 7-upla $(\Sigma, \Gamma, Q, \delta, \mathfrak{b}, q_0, F)$, donde Σ, Γ, Q, $\mathfrak{b}$, q_0 y F son como en la definición 2.43 y δ es una función tal que $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{-1, 0, 1\}^k$.

Como es de esperarse, $\delta(q, r_1, \ldots, r_k) = (q', s_1, \ldots, s_k, \lambda_1, \ldots, \lambda_k)$ significa que la máquina al estar en estado q y leer r_i en la i-ésima cinta, pasa a estado q', escribe s_i en la i-ésima cinta y mueve cada cabezal a la derecha o a la izquierda si $\lambda_i \in \{-1, 1\}$, o no mueve el cabezal, si $\lambda_i = 0$.

Observación 2.57. En la definición 2.56, es necesario que el rango de δ esté contenido $Q \times \Gamma^k \times \{-1, 0, 1\}^k$ y no en $Q \times \Gamma^k \times \{-1, 1\}^k$. De no ser así, la máquina tendría la seria limitación de no poder mantener fijo un cabezal y mover otro. Este particular movimiento de cabezales no puede ser simulado si cada cabezal debe moverse obligatoriamente.

Es posible definir análogos para una máquina de Turing de múltiples cintas de las definiciones 2.44 – 2.47. En este caso, sólo definiremos explícitamente la noción de *descripción instantánea*.

Definición 2.58. Sea M una máquina de Turing de k cintas. Una **descripción instantánea** de M es una cadena de la forma

$$C = q|x_1 \cdot y_1|x_2 \cdot y_2 \ldots |x_k \cdot y_k$$

donde suponemos que los caracteres | y · no pertenecen a Q ni a Γ, $q \in Q$, $x_1, \ldots, x_k, y_1, \ldots, y_k \in \Gamma^*$ y $y_i \neq \epsilon$, para todo $i = 1, \ldots, k$. En este caso, una descripción instantánea de M se dirá **inicial** si es de la forma

$$C = q|\epsilon \cdot w|\epsilon \cdot \flat \ldots |\epsilon \cdot \flat,$$

y se dirá **final** si es de la forma $C = q|x_1 \cdot y_1|x_2 \cdot y_2 \ldots |x_k \cdot y_k$, donde $q \in \{q_a, q_r\}$.

Observe que la definición 2.44 no es el caso particular para $k = 1$ de la definición 2.58. Para $k = 1$, una descripción instantánea (según la definición 2.58) es de la forma $q|x \cdot y$. Es fácil ver que ambas son equivalentes, sin embargo, en adelante usaremos la definición 2.58 sólo para máquinas de $k \geqslant 2$ cintas.

Ejemplo 2.59. Consideremos el problema de determinar si dos cadenas del alfabeto $\{0\}$ tienen el mismo tamaño, como vimos en el ejemplo 2.51. Para resolverlo, construiremos una máquina de múltiples cintas $M = (\Sigma, \Gamma, Q, \delta, \flat, q_0, F)$, donde

1. $\Sigma = \{0, |\}$.
2. $\Gamma = \Sigma \cup \{\flat\}$.
3. $F = \{q_s, q_n\}$.
4. $Q = \{q_0, q_1\} \cup F$.
5. δ está definida de la siguiente manera

δ	$(0, \flat)$	$(\flat, 0)$	$(0, 0)$	$(\|, \flat)$	$(\flat, \flat)$
q_0	$(q_0, \flat, 0, 1, 1)$			$(q_1, \flat, \flat, 1, -1)$	
q_1	$(q_n, \flat, 0, 1, 1)$	$(q_n, \flat, 0, 1, 1)$	$(q_1, \flat, \flat, 1, -1)$		$(q_s, \flat, \flat, -1, -1)$

Como en el ejemplo 2.51, M acepta entradas de la forma $0^n|0^m$ y devuelve q_s, si $n = m$; o q_n en caso contrario. La máquina M funciona de la siguiente manera: el estado q_0 copia la primera cadena de ceros en la segunda cinta. Apenas el cabezal de la primera cinta encuentra |, posiciona el cabezal al inicio de la segunda cadena en la primera cinta y al final de la primera cadena en la segunda cinta, y pasa a estado q_1. En estado q_1 recorre ambas cintas mientras encuentra 0 en ambas. La máquina termina apenas deje de encontrar 0 escrito en ambas cintas a la vez. En este caso, si encuentra ambas casillas vacías termina en estado q_s. En caso contrario, termina en estado q_n.

Puede parecer que una máquina de Turing de múltiples cintas puede realizar tareas que una máquina de Turing común no podría. Sin embargo, este no es el caso: probaremos que toda máquina de Turing de múltiples cintas puede ser *simulada* por una máquina de Turing de una cinta. Una máquina M **simula** una máquina M' si M y M' tienen el mismo alfabeto de entrada y, para cualquier entrada w, M y M' tienen la misma salida.

Luego, tenemos el siguiente teorema.

Teorema 2.60. *Toda máquina de Turing de $k \geqslant 2$ cintas puede ser simulada por una máquina de Turing de $k-1$ cintas.*

Demostración: Sea $M = (\Sigma, \Gamma, Q, \delta, \flat, q_0, F)$ una máquina de Turing de $k \geqslant 2$ cintas. Mostraremos que podemos simular la ejecución de M con una máquina de Turing M' de $k-1$ cintas. La idea es simular las cintas $k-1$ y k de M en la cinta $k-1$ de M'.

Definamos el alfabeto $\dot{\Gamma} = \left\{\dot{r} : r \in \Gamma\right\}$ que, sin pérdida de generalidad, podemos asumir disjunto de $Q \cup \Gamma$. Dada una descripción instantánea C de M construiremos una "descripción instantánea"[2] C' de M' que represente biunívocamente C.

Sea $C = q|x_1 \cdot y_1|x_2 \cdot y_2| \cdots |x_{k-1} \cdot y_{k-1}|x_k \cdot y_k$. Como $y_{k-1}, y_k \neq \epsilon$, podemos escribir $y_{k-1} = r_{k-1}z_{k-1}$ y $y_k = r_k z_k$, donde $r_{k-1}, r_k \in \Gamma$ y $z_{k-1}, z_k \in \Gamma^*$. Definamos C' como

$$C' = q|x_1 \cdot y_1|x_2 \cdot y_2| \cdots |x_{k-1} \cdot \dot{r}_{k-1}\, z_{k-1} \# x_k\, \dot{r}_k\, z_k$$

Observe que usamos los caracteres de $\dot{\Gamma}$ para denotar que el cabezal las cintas $k-1$ y k de M apuntan al caracter correspondiente. En este caso, separaremos el contenido de las cintas $k-1$ y k de M usando el caracter $\#$, que supondremos no está en Γ.

En caso que C sea una descripción instantánea inicial de M, entonces tenemos

$$\begin{aligned} C' &= q|\epsilon \cdot w|\epsilon \cdot \flat \cdots |\epsilon \cdot \dot{\flat}\, \epsilon \# \epsilon\, \dot{\flat}\, \epsilon \\ &= q|\cdot w| \cdot \flat \cdots |\cdot \dot{\flat} \# \dot{\flat}, \end{aligned} \tag{2.2}$$

donde w es la entrada de la máquina M.

Bastará definir entonces una máquina de Turing M' de $k-1$ cintas tal que, dadas dos descripciones instantáneas C_1 y C_2 de M tales que $C_1 \overset{M}{\longmapsto} C_2$, podamos convertir C_1' en C_2' vía ejecuciones de M', es decir, se cumpla

$$C_1' \overset{M'}{\longmapsto} \cdots \overset{M'}{\longmapsto} C_2'.$$

Sean C_1 y C_2 dos descripciones instantáneas de M tales que $C_1 \overset{M}{\longmapsto} C_2$. Escribamos

$$C_1 = q|x_1 \cdot r_1 z_1|x_2 \cdot r_2 z_2| \cdots |x_{k-1} \cdot r_{k-1} z_{k-1}|x_k \cdot r_k z_k$$

[2]Observe que no hemos aún definido M', así que sólo construiremos una cadena C'

y supongamos que $\delta(q, r_1, \ldots, r_k) = (\overline{q}, s_1, \ldots, s_k, \lambda_1, \ldots, \lambda_k)$. Definamos una máquina de Turing M', con alfabeto $\Gamma' = \Gamma \cup \dot{\Gamma} \cup \{\#\}$ y alfabeto de entrada Σ, tal que, teniendo una descripción instantánea

$$C_1' = q|x_1 \cdot r_1 z_1|x_2 \cdot r_2 z_2| \cdots |x_{k-1} \cdot \dot{r}_{k-1}\, z_{k-1} \# x_k\, \dot{r}_k\, z_k,$$

convierta C_1' a C_2' de la siguiente manera:

1. M' sólo usará estados pertenecientes a cierto conjunto finito Q_q, que depende del estado q de C_1. Esto se hace para que M' "recuerde" el estado original q de C_1. Para no caer en ambigüedades, impondremos que $Q_{q_1} \cap Q_{q_2} = \varnothing$ si $q_1 \neq q_2$.

2. En la primera ejecución de M' teniendo a C_1' como descripción instantánea, convertimos las $k-2$ primeras cintas tal como lo haría M. Luego, en las siguientes ejecuciones, no modificamos tales cintas ni la posición del cabezal en ellas.

3. Observe que en la primera ejecución de M' teniendo a C_1' como descripción instantánea, el cabezal de la cinta $k-1$ de M' apunta a un caracter de $\dot{\Gamma}$. En este caso, convertir tal caracter en su correspondiente en Γ, luego convertirlo tal como lo haría M en su cinta $k-1$. Una vez hecho esto, convertir el caracter al que apunta el cabezal en su correspondiente en $\dot{\Gamma}$. En caso tal caracter sea $\#$, M' deberá mover todo el contenido a la derecha de $\#$ de la cinta $k-1$ una posición a la derecha y escribir $\dot{\flat}$ donde se encontraba $\#$.

   Así, hemos simulado la acción de M en su cinta $k-1$.

4. Movemos el cabezal de M' a la derecha hasta encontrar el caracter de $\dot{\Gamma}$ más a la derecha de la cinta $k-1$ de M'. Claramente, este caracter no es el caracter recién escrito en el paso anterior.

5. Convertimos el caracter apuntado por el cabezal de la cinta $k-1$ de M' en su correspondiente en Γ, luego convertiremos tal como lo haría M en su cinta k. Una vez hecho esto, convertimos el caracter al que apunta el cabezal en su correspondiente en $\dot{\Gamma}$. En caso tal caracter sea $\#$, M' deberá mover todo el contenido a la izquierda de $\#$ de la cinta $k-1$ una posición a la izquierda y escribir $\dot{\flat}$ donde se encontraba $\#$.

   Así, hemos simulado la acción de M en su cinta k.

6. Finalmente, movemos el cabezal de la cinta $k-1$ de M' hasta el caracter de $\dot{\Gamma}$ más a la izquierda de la cinta $k-1$. Claramente, este caracter no es el caracter recién escrito en el paso anterior. Una vez hecho esto, pasar al estado $\overline{q}$.

Además, al empezar la ejecución de M' con entrada w, el cabezal de la cinta $k-1$ deberá escribir $\dot{\flat}\ \#\ \dot{\flat}$ y posicionarse encima del primer $\dot{\flat}$ y pasar a estado q_0. Así, M' convierte su descripción instantánea inicial en una descripción instantánea inicial de M, como en (2.2).

Así, dada una ejecución $(C_k)_{k\in I}$ de M, con entrada w, existe una ejecución de M':

$$\overline{C}_0 \overset{M'}{\longmapsto} \cdots \overset{M'}{\longmapsto} C'_0 \overset{M'}{\longmapsto} \cdots \overset{M'}{\longmapsto} C'_1 \cdots C'_k \overset{M'}{\longmapsto} \cdots \overset{M'}{\longmapsto} C'_{k+1} \cdots,$$

que es infinita en caso $(C_k)_{k\in I}$ sea infinita; o es finita y tiene a C'_m como descripción instantánea final, en caso $(C_k)_{k\in I}$ sea finita y $\#I = m$. Esto implica que M y M' tienen la misma salida y por lo tanto M' simula M. □

Del teorema anterior se desprende el siguiente corolario.

Corolario 2.61. *Toda máquina de Turing de $k \geqslant 2$ cintas puede ser simulada por una máquina de Turing de una cinta.*

Demostración: Basta considerar el caso $k = 2$. La única modificación que hay que hacer a M' en este caso es el de borrar el contenido simulado de la segunda cinta de la máquina M al finalizar la ejecución simulada de esta. Se hace esto para que M y M' tengan la misma salida. □

Observación 2.62. En el teorema 2.60 pudimos haber descrito M' como una máquina de una cinta. En este caso, tendríamos que repetir el paso 3 para que M' simule la ejecución de M en cada una de las $k-1$ cintas restantes.

Finalmente, definimos la noción de *Turing computable*.

Definición 2.63. Diremos que una función $f : \mathbb{N}^r \to \mathbb{N}$ es **Turing computable** si puede ser calculado usando una máquina de Turing, esto es, si existe una máquina de Turing M que, con entrada $\overline{m} \in \mathbb{N}^r$, codificada en algún alfabeto adecuado, se detenga y devuelva $f(\overline{m})$ en su cinta.

2.5. La tesis de Church-Turing

Las nociones de λ-cálculo, recursividad y maquinas de Turing que vimos en las secciones anteriores sirven para dar diferentes versiones de *computabilidad* (definiciones 2.24, 2.38 y 2.63).

Las nociones de λ-calculable y recursividad fueron probadas equivalentes en [Kleene, 1936], pero ya había sido mencionado por Church en [Church, 1936]. Esto motivó a Church a hacer la siguiente afirmación:

> *El hecho, sin embargo, que dos ampliamente diferentes y (en opinión del autor) igualmente naturales definiciones de efectivamente calculable resulten ser equivalentes añade fuerza a los razonamientos aducidos antes para creer que ellas constituyen una caracterización tan general de esta noción como consistente con el entendimiento intuitivo usual de esta.*

También en 1936, Alan Turing en [Turing, 1936] definió el concepto de *máquina* y probó que, de hecho, tal concepto es equivalente al λ-cálculo dado por Church. Además es en [Turing, 1936] donde Turing introduce la noción de *función computable* como una función que puede ser calculada por una máquina[3].

Así, la tesis de Church afirma que las definiciones 2.4, 2.24, 2.38 y 2.63 son equivalentes. Dicho de otra manera:

> *una función computable es aquella que puede ser calculada usando indistintamente máquinas de Turing, o λ-cálculo o funciones recursivas.*

En general, saliendo del contexto de las funciones naturales, la tesis de Church afirma que la noción informal de algoritmo puede ser formalizada usando, por ejemplo, máquinas de Turing. Esta afirmación es fundamental en la teoría de la computación, pues permite hacer un tratamiento formal de algoritmos.

2.6. Una función no computable: el problema del castor ocupado

En esta sección presentaremos un ejemplo de función no computable. Este ejemplo fue dado por el matemático húngaro Tibor Radó en 1962 [Radó, 1962]. Haremos uso de una variante de máquina de Turing que permite iniciar la ejecución sin ninguna entrada escrita en su cinta.

Dado $n \in \mathbb{N}$, el **juego del castor ocupado de n estados** (en ingles: *n-state busy beaver game*), y lo abreviaremos como **juego BB-n**, es una competencia donde participan máquinas de Turing de una cinta tales que

1. su alfabeto es el conjunto $\{\flat, 1\}$;
2. su conjunto de estados es dado por $\{q_0, \ldots, q_n, q_{n+1}\}$, donde q_0 es el estado inicial y q_{n+1} es el único estado final; y
3. con entrada vacía, la máquina se detiene.

El **puntaje** de una máquina de Turing participante es la cantidad de símbolos 1 escritos en su cinta al final de su ejecución. La competencia consiste en encontrar la máquina de máximo puntaje.

Definición 2.64. Diremos que una máquina de Turing, participante del juego BB-n, es un **castor ocupado de n estados** (en ingles *n-state busy beaver*), y lo abreviaremos como **BB-n**, si es la máquina de mayor puntaje entre todas las máquinas participantes.

En general, para máquinas de r estados no finales y s símbolos, para cada estado q no final y símbolo a, la cantidad de posibles transiciones $\delta(q, a) = (q', a', \lambda')$ es $r \times s \times 2 = 2rs$. Por lo tanto, la cantidad

[3]de Turing

total de funciones de transición es $(2rs)^{(r-f)s}$, donde f es la cantidad de estados finales de la máquina. Así, para maquinas competidoras del juego BB-n, existen

$$(4(n+1))^{2n}$$

posibles funciones de transición. Esto es, existen a lo más $(4(n+1))^{2n}$ máquinas competidoras del juego BB-n. Esto implica que siempre existe un BB-n, para cada $n \in \mathbb{N}$.

Defina ahora E_n como el conjunto de máquinas de Turing competidoras de BB-n. Es claro que E_n es no vacío y, por lo visto anteriormente, $\#E_n \leqslant (4(n+1))^{2n}$, luego E_n es finito. Ahora, para $M \in E_n$, denotemos por $\sigma(M)$ al puntaje de M. Así, la función $\Sigma : \mathbb{N} \to \mathbb{N}$ dada por

$$\Sigma(n) = \text{máx}\{\sigma(M) \ : \ M \in E_n\},$$

está bien definida, para cada $n \in \mathbb{N}$. Llamaremos Σ como **función del castor ocupado**.

Teorema 2.65. *La función $\Sigma : \mathbb{N} \to \mathbb{N}$ no es Turing computable.*

Demostración: Sea $f : \mathbb{N} \to \mathbb{N}$ computable, probaremos que $\Sigma(n) > f(n)$, para todo n suficientemente grande. Definamos la función $F : \mathbb{N} \to \mathbb{N}$ definida por

$$F(n) = \sum_{k=1}^{n} (f(k) + k^2).$$

Claramente tenemos

$$F(n) \geqslant f(n), \tag{2.3}$$

$$F(n) \geqslant n^2, \tag{2.4}$$

$$F(n+1) > F(n), \tag{2.5}$$

para todo $n \in \mathbb{N}$. Además, como f es computable entonces F también lo es. Luego existe una máquina de Turing M_F que computa F. Sin perdida de generalidad, podemos suponer que M_F lee n símbolos 1, los borra y escribe $F(n)$ símbolos 1 consecutivos, posicionando el cabezal en el 1 mas a la izquierda. Supongamos además que M_F tiene r estados. Por otro lado, dado $m \in \mathbb{N}$, sea M_m una máquina de Turing con $m+1$ estados que en una cinta en blanco escribe m símbolos 1 y posiciona el cabezal encima del 1 más a la izquierda.

Considere la máquina $N_{F,m}$ resultante de la *composición* $M_F \circ M_F \circ M_m$. Esto es, M_m escribe m símbolos 1, luego evalúa M_F en el resultado y luego evalúa M_F nuevamente. No es difícil convencerse de que $N_{F,m}$ tiene $m+1+2r$ estados. Así $N_{F,m}$ es una máquina competidora del juego BB-$(m+1+2r)$, con un puntaje de

$$\sigma(N_{F,m}) = F(F(m)).$$

Luego, tenemos la desigualdad

$$\Sigma(1 + m + 2r) \geqslant F(F(m)).$$

Para todo m suficientemente grande, $m^2 > 1 + m + 2r$ y usando (2.4) tenemos

$$F(m) > 1 + m + 2r.$$

Y, por (2.3) y (2.5),

$$\Sigma(1 + m + 2r) \geqslant F(F(m)) > F(1 + m + 2r) > f(1 + m + 2r).$$

Esto implica que $\Sigma(n) > f(n)$, para n suficientemente grande, y tenemos el teorema demostrado. □

Capítulo 3

Decidibilidad

3.1. Máquinas de decisión y lenguajes decidibles

Un caso particular de máquinas de Turing será de nuestro especial interés. En adelante, usaremos máquinas de Turing que determinen (o dicho de otra manera, *decidan*) si su entrada pertenece a un conjunto dado.

Definición 3.1. Se dice que una máquina de Turing M es de **decisión** si el conjunto de estados finales es de la forma $F = \{q_a, q_r\}$. El estado q_a se denominará **estado de aceptación** y el estado q_r se denominará **estado de rechazo**.

Observaciones.

1. Es común en la literatura (vea por ejemplo [Cook, 2006, Lucchesi et al., 1979, Sipser, 1996]) definir directamente las máquinas de Turing como máquinas de Turing de decisión. En este caso una máquina de Turing de decisión sería una 8-upla $(\Sigma, \Gamma, Q, \delta, \flat, q_0, q_a, q_r)$. Las definiciones de "descripción instantánea" y "ejecución" permanecen siendo las mismas, al igual que las interpretaciones de cada uno de los elementos de la máquina de Turing.

2. Para el caso de máquinas de Turing de decisión, no tomaremos en cuenta el contenido final de su cinta y nos concentraremos en el estado final de su cabezal.

En lo que sigue, salvo indiquemos lo contrario, usaremos sólo máquinas de Turing de decisión.

Definición 3.2. Sea M una máquina de Turing. Diremos que M **acepta** w si, con entrada w, la máquina M se detiene y termina en estado final q_a. Diremos que M **rechaza** w si, con entrada w, la máquina se detiene y termina en estado final q_r.

Observe que una máquina M podría no detenerse para alguna entrada w, pero en caso se detenga, siempre determinará si acepta o rechaza w.

Definición 3.3. Sea M una máquina de Turing. El **lenguaje de aceptación** de M es el conjunto de entradas $w \in \Sigma^*$ aceptadas por M, es decir

$$L_M = \{w \in \Sigma^* \ : \ M \text{ acepta } w\}.$$

Ahora consideremos el problema inverso, es decir, dado un lenguaje L sobre cierto alfabeto Σ, ¿existirá una máquina de Turing M tal que $L = L_M$? Una respuesta afirmativa a esta pregunta implicaría que para cualquier $w \in \Sigma^*$ candidato a estar en L, la máquina M nos confirmaría que efectivamente $w \in L$.

Definición 3.4. Diremos que un lenguaje L sobre Σ es **reconocible** o **semidecidible**, si existe M tal que $L = L_M$, es decir, $w \in L$ si, y sólo si, w es aceptado por M. En este caso diremos que M **reconoce** L.

Sin embargo, si $w \notin L_M$ no significa que w es rechazado por M, pues M bien puede no detenerse con entrada w. En la práctica, es difícil[1] distinguir si una máquina con entrada w no se detiene, o bien está tomando un tiempo largo en detenerse. Es claro que queremos evitar este tipo de situaciones.

Definición 3.5. Diremos que un lenguaje L sobre Σ es **decidible**, si $L = L_M$ para alguna máquina M con la propiedad de que M se detiene para toda entrada $w \in \Sigma^*$. En este caso diremos que M **decide** L.

Observación 3.6. Es también estándar (cf. [Hopcroft et al., 2001, Lewis and Papadimitriou, 1997]) llamar **recursivamente numerable** a un lenguaje semidecidible, y **recursivo** a un lenguaje decidible.

Ejemplo 3.7. Sea $\Sigma = \{0, 1, |\}$ y definamos el lenguaje

$$L = \{m_{(2)} | n_{(2)} \in \Sigma^* \ : \ m, n \in \mathbb{N},\ m \leqslant n\}.$$

Construiremos una máquina M que decide L. Por el corolario 2.61, podemos construir M de 2 cintas y procederemos como en el ejemplo 2.59. Sea $M = (\Sigma, \Gamma, Q, \delta, \flat, q_0, q_a, q_r)$ de 2 cintas, donde

1. $\Sigma = \{0, 1, |\}$.
2. $\Gamma = \Sigma \cup \{\flat\}$.
3. $Q = \{q_0, q_a, q_r, q_1\}$.
4. δ está definida de la siguiente manera

δ	$(r, \flat)$	$(\flat, s)$	(r, s)	$(\|, \flat)$	$(\flat, \flat)$
q_0	$(q_0, \flat, r, 1, 1)$			$(q_1, \flat, \flat, 1, -1)$	
q_1	$(q_a, r, \flat, 1, 1)$	$(q_r, \flat, s, 1, 1)$	$(q_1, r, s, 1, -1)$		$(q_2, \flat, \flat, -1, 1)$
q_2		$(q_3, \flat, s, 1, 0)$	$(q_2, r, s, -1, 0)$		

[1] En general es imposible, vea la sección 3.2.3.

donde $r, s = 0, 1$, y δ en estado q_3 está definido por

δ	$(1,1)$	$(0,0)$	$(\flat,\flat)$	$(1,0)$	$(0,1)$
q_3	$(q_3,1,1,1,1)$	$(q_3,0,0,1,1)$	$(q_a,\flat,\flat,-1,-1)$	$(q_a,1,0,1,1)$	$(q_r,0,1,1,1)$

Sean $m, n \in \mathbb{N}$ y suponga que ejecutamos M con entrada $w = m_{(2)}|n_{(2)}$. En este caso, M funciona de la siguiente manera: los estados q_0 y q_1 son análogos a los respectivos estados de la máquina definida en el ejemplo 2.59. Estos sirven para comparar el tamaño de $m_{(2)}$ y $n_{(2)}$. La máquina acepta la entrada si encuentra que la longitud de $m_{(2)}$ es estrictamente menor que la longitud de $n_{(2)}$, y la rechaza si ocurre lo opuesto. Al igual que en el ejemplo 2.59, M copia $m_{(2)}$ en su segunda cinta y borra el caracter |. En caso M detecte que $m_{(2)}$ y $n_{(2)}$ tienen la misma longitud, entonces pasa a estado q_2, posiciona los cabezales de la primera cinta y segunda cinta al inicio de $m_{(2)}$ y $n_{(2)}$, respectivamente, y luego pasa a estado q_3. Finalmente, el estado q_3 se encarga de comparar paralelamente los dígitos de $m_{(2)}$ y $n_{(2)}$ hasta encontrar una discrepancia. De no encontrarla, acepta la entrada (pues $m = n$). Por otro lado, si encuentra $(1,0)$ significa que $n > m$ y acepta la entrada, y rechaza la entrada de encontrar $(0,1)$.

Ejemplo 3.8. Sobre $\Sigma = \{0,1\}$ consideremos el lenguaje de las cadenas palíndromas

$$L = \{a_1 \cdots a_n \in \Sigma^* \ : \ a_i = a_{n-i+1} \text{ para todo } i = 1, \ldots, n\} \cup \{\epsilon\}$$

Una máquina que decide L procede de la siguiente manera:

- Elimina el primer caracter de la entrada.
- Mueve el cabezal hasta el último caracter.
- Si el último caracter es igual al primero, elimina dicho caracter y mueve el cabezal hasta el (nuevo) primer caracter.
- Repite lo anterior hasta no quedar más caracteres o encontrar que el último caracter no es igual al primero.

Considere la máquina de Turing $M = (\Sigma, \Gamma, Q, \delta, \flat, q_0, q_a, q_r)$ tal que

1. $\Gamma = \{0,1,\flat\}$;
2. $\Sigma = \{0,1\}$;
3. $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_a, q_r\}$;

4. δ está definida de la siguiente manera:

δ	0	1	b
q_0	$(q_1, \mathtt{b}, 1)$	$(q_4, \mathtt{b}, 1)$	$(q_a, \mathtt{b}, 1)$
q_1	$(q_1, 0, 1)$	$(q_1, 1, 1)$	$(q_2, \mathtt{b}, -1)$
q_2	$(q_3, \mathtt{b}, -1)$	$(q_r, 1, -1)$	$(q_a, \mathtt{b}, -1)$
q_3	$(q_3, 0, -1)$	$(q_3, 1, -1)$	$(q_0, \mathtt{b}, 1)$
q_4	$(q_4, 0, 1)$	$(q_4, 1, 1)$	$(q_5, \mathtt{b}, -1)$
q_5	$(q_r, \mathtt{b}, -1)$	$(q_3, \mathtt{b}, -1)$	$(q_a, \mathtt{b}, -1)$

En este caso, el estado q_0 lee el primer caracter de la cadena y pasa a estado q_1, en caso este sea 0, o a estado q_4, en caso este sea 1. Los estados q_1 y q_4 simplemente mueven el cabezal de la máquina hasta el último caracter y luego cambian al estado q_2 y q_5 respectivamente. El estado q_2 cambia al estado de rechazo q_r si encuentra un 1, o borra el contenido y pasa a estado q_3 en caso encuentre 0, o acepta la entrada en caso encuentre b. Análogamente, q_4 cambia a q_r si encuentra un 0, acepta si encuentra b o borra y pasa a q_3 si encuentra un 1. El estado q_3 simplemente posiciona el cabezal en el (nuevo) primer elemento de la cadena y cambia a estado q_0, y todo se repite hasta que el cabezal en estado q_0 encuentre b.

3.2. Máquina de Turing universal y lenguajes no decidibles

Si tuviéramos que dar un análogo en la vida real de las máquinas de Turing, entonces una máquina de Turing sería una pequeña computadora cuyos circuitos contienen las instrucciones de funcionamiento (la función de transición), el disco duro almacenaría los datos que son procesados (la cinta) y el CPU se encargaría de ejecutar las acciones (el cabezal). Además tendría un teclado, adecuado para ingresar caracteres de un alfabeto Σ, y un monitor, capaz de mostrar caracteres de cierto alfabeto Γ y/o las palabras `acepta` y `rechaza`.

Cada una de estas computadoras tiene incorporado un único programa, a diferencia de las computadoras de uso común, que tienen un "gran programa" (sistema operativo) que se encarga de ejecutar programas más pequeños a voluntad del usuario. La máquina de Turing universal es el análogo a un *sistema operativo* de las máquinas de Turing, pues ejecuta máquinas de Turing con alguna entrada dada.

Observe que esto no es lo mismo que lo hecho en la demostración del teorema 2.60, donde se construyó una máquina M' que dependía de otra máquina M. En este caso, la máquina de Turing universal U tiene instrucciones especificas tales que, dada cualquier máquina M, codificada de alguna manera, y una entrada w de M, U literalmente ejecuta M con entrada w y devuelve la salida de M.

Es así que necesitamos una manera de codificar la información de una máquina de Turing, de tal manera que cualquier máquina de Turing pueda ser representada, sin ambigüedad, por una cadena en algún alfabeto.

3.2.1. Codificación de máquinas de Turing

Sea $M = (\Sigma, \Gamma, Q, \delta, \flat, q_0, q_a, q_r)$ una máquina de Turing. Sin pérdida de generalidad, podemos reemplazar Γ por cualquier conjunto finito de la misma cardinalidad, de acuerdo a nuestra conveniencia, modificando adecuadamente Σ, δ y $\flat$. De la misma manera, podemos reemplazar Q por cualquier conjunto finito con la misma cardinalidad, siempre y cuando no interseque el alfabeto de la máquina. Así, supongamos que $\Gamma = \{1, \ldots, n\}$, $\Sigma = \{1, \ldots, r\}$, con $r < n$, y $Q = \{\bar{0}, \bar{1}, \bar{2}, \ldots, \bar{s}\}$. En este caso, definimos $\flat = n$, $q_0 = \bar{0}$, $q_a = \bar{1}$ y $q_r = \bar{2}$. Por lo tanto, M tiene la forma

$$\big(\{1, \ldots, r\}, \{1, \ldots, n\}, \{\bar{0}, \ldots, \bar{s}\}, \delta, n, \bar{0}, \bar{1}, \bar{2}\big). \tag{3.1}$$

Vamos a asociar, a cada elemento de Γ y Q, una palabra en el alfabeto $\{0, 1, \mathtt{s}, \mathtt{e}\}$. Entonces, a cada $p \in \Gamma$ y $\bar{q} \in Q$, les asociamos las palabras $\mathtt{s}p_{(2)}$ y $\mathtt{e}q_{(2)}$, respectivamente. De esto, podemos dar la siguiente definición.

Definición 3.9. Dado $w = w_1 \cdots w_k \in \Gamma^*$, la **codificación** de w es la cadena $\langle w \rangle = \mathtt{e}w_{1(2)} \cdots \mathtt{e}w_{k(2)}$.

Ejemplo 3.10. Sea $\Gamma = \{1, \ldots, 100\}$ y $Q = \{\bar{0}, \ldots, \overline{200}\}$. Entonces $47 \in \Gamma$ se codifica como $\mathtt{s}101111$ y $\overline{148} \in Q$ se codifica como $\mathtt{e}10010100$. Además, en este caso, $\flat$, q_0, q_a y q_r se codifican como $\mathtt{s}1100100$, $\mathtt{e}0$, $\mathtt{e}1$ y $\mathtt{e}10$, respectivamente.

Ejemplo 3.11. Consideremos la máquina de Turing M definida en el ejemplo 3.8. Por lo dicho anteriormente, podemos suponer que $\Sigma = \{1, 2\}$, $\Gamma = \Sigma \cup \{3\}$ y $Q = \{\bar{0}, \bar{1}, \bar{2}, \bar{3}, \ldots, \bar{7}\}$, donde $\flat = 3$, $q_a = \bar{1}$ y $q_r = \bar{2}$. En este caso, codificando los elementos de Γ y Q obtenemos: $\{\mathtt{s}1, \mathtt{s}10, \mathtt{s}11\}$ y $\{\mathtt{e}0, \mathtt{e}1, \mathtt{e}10, \mathtt{e}11, \mathtt{e}100, \mathtt{e}101, \mathtt{e}110, \mathtt{e}111\}$, respectivamente.

Ahora asociemos una codificación a δ. A cada transición $\delta(q, r) = (q', r', \lambda)$ le asociamos el código

$$\mathtt{e}q_{(2)}\mathtt{s}r_{(2)}\mathtt{e}q'_{(2)}\mathtt{s}r'_{(2)}\mathtt{m}\hat{\lambda} \in \{0, 1, \mathtt{s}, \mathtt{e}, \mathtt{m}\}^*,$$

donde $\hat{\lambda} = 0$ si $\lambda = -1$ y $\hat{\lambda} = 1$ si $\lambda = 1$. Observe que el código anterior representa sin ambigüedad a la transición dada inicialmente. Así, la función de transición δ es codificada por la concatenación de todas las codificaciones de sus transiciones. En general la cadena resultante no es única, sin embargo, esto no será inconveniente pues existen solo finitas de ellas. Denotaremos a esta cadena como $\langle \delta \rangle$.

Ejemplo 3.12 (continuación del ejemplo 3.11). En este caso, la función de transición δ está dada por

δ	0	1	b
q_0	$(q_1, \mathtt{b}, 1)$	$(q_4, \mathtt{b}, 1)$	$(q_a, \mathtt{b}, 1)$
q_1	$(q_1, 0, 1)$	$(q_1, 1, 1)$	$(q_2, \mathtt{b}, -1)$
q_2	$(q_3, \mathtt{b}, -1)$	$(q_r, 1, -1)$	$(q_a, \mathtt{b}, -1)$
q_3	$(q_3, 0, -1)$	$(q_3, 1, -1)$	$(q_0, \mathtt{b}, 1)$
q_4	$(q_4, 0, 1)$	$(q_4, 1, 1)$	$(q_5, \mathtt{b}, -1)$
q_5	$(q_r, \mathtt{b}, -1)$	$(q_3, \mathtt{b}, -1)$	$(q_a, \mathtt{b}, -1)$

de donde, codificando cada transición, obtenemos

δ	s1	s10	s11
e0	e0s1e11s11m1	e0s10e110s11m1	e0s11e1s11m1
e11	e11s1e11s1m1	e11s10e11s10m1	e11s11e100s11m0
e100	e100s1e101s11m0	e100s10e10s10m0	e100s11e1s11m0
e101	e101s1e101s1m0	e101s10e101s10m0	e101s11e0s11m1
e110	e110s1e110s1m1	e110s10e110s10m1	e110s11e111s11m0
e111	e111s1e10s11m0	e111s10e101s11m0	e111s11e1s11m0

y así, la codificación $\langle\delta\rangle$ de δ sería

e0s1e11s11m1e0s10e110s11m1e0s11e1s11m1e11s1e11s1m1
e11s10e11s10m1e11s11e100s11m0e100s1e101s11m0e100s10e10s10m0
e100s11e1s11m0e101s1e101s1m0e101s10e101s10m0e101s11e0s11m1
e110s1e110s1m1e110s10e110s10m1e110s11e111s11m0e111s1e10s11m0
e111s10e101s11m0e111s11e1s11m0.

Observe que del código anterior podemos recuperar la función δ original, respetando la convención de que $\mathtt{b} = \mathtt{s11}$, $q_0 = \mathtt{e0}$, $q_a = \mathtt{e1}$ y $q_r = \mathtt{e10}$.

Como ya mencionamos, la codificación de la función de transición δ de una máquina M no es única, sin embargo, de cualquier codificación de δ podemos siempre recuperar δ sin ambigüedades. De hecho, la codificación de δ almacena toda la información necesaria para, suponiendo ya definida la máquina de Turing universal, ejecutar M con alguna entrada. Sin embargo, dada una cadena $C = \langle\delta\rangle$ que representa una función de transición δ de alguna máquina M, no podemos recuperar M sin ambigüedades. Esto se debe a que C no almacena información acerca de Σ, Γ y Q.

Definición 3.13. Sea $M = (\{1, \ldots, r\}, \{1, \ldots, n\}, \{\bar{0}, \ldots, \bar{s}\}, \delta, n, \bar{0}, \bar{1}, \bar{2})$ una máquina de Turing. Una

codificación de M es una cadena de la forma

$$\langle M \rangle = r_{(2)}|n_{(2)}|s_{(2)}\#\langle\delta\rangle,$$

sobre el alfabeto $\mathcal{C} = \{0, 1, \mathtt{s}, \mathtt{e}, \mathtt{m}, \#, |\}$, donde $\langle\delta\rangle$ es alguna codificación de la función de transición δ.

Observe cómo una codificación de una máquina M guarda similitud con el código de un programa. El símbolo # divide a $\langle M \rangle$ en dos partes, $r_{(2)}|n_{(2)}|s_{(2)}$ y $\langle\delta\rangle$. La primera cadena es análoga a la *cabecera* de un programa, y la segunda denota las *instrucciones* del mismo.

Previo a la definición de máquina de Turing universal, construiremos una máquina de Turing que decide codificaciones de máquinas de Turing.

Proposición 3.14. *El lenguaje* $\mathfrak{L}_{TM} = \{\langle M \rangle \in \mathcal{C}^* \ : \ M \text{ es máquina de Turing}\}$ *es decidible.*

Demostración: Usaremos una máquina de Turing M_R de 5 cintas con alfabeto de entrada $\mathcal{C}$. Inicialmente, M_R moverá $r_{(2)}$, $n_{(2)}$ y $s_{(2)}$ a la segunda, tercera y cuarta cinta, respectivamente. En caso algo impida a M_R completar la tarea (por ejemplo, encontrar $\flat$ o # donde no debería) rechazar la entrada. Al terminar, la primera cinta deberá contener sólo $\langle\delta\rangle$ y tener los cabezales apuntando el inicio de las cadenas escritas en todas sus cintas. Luego, M_R decidirá si $r < n$. Esto puede ser conseguido adaptando adecuadamente la máquina definida en el ejemplo 3.7. En caso no se tenga $r < n$, rechazar la entrada.

Ahora M_R deberá decidir si la cadena escrita en la primera cinta codifica una función de transición. Debido a la definición de $\langle\delta\rangle$, bastará saber decidir la codificación de una transición. Primero M_R moverá la primera transición que encuentre en su primera cinta a la quinta cinta de M_R. Para esto, M_R moverá caracter por caracter de la primera a la quinta cinta, hasta encontrar el símbolo m. Al encontrar m, M_R moverá este y el siguiente caracter a la quinta cinta. En caso M_R encuentre un símbolo diferente de $0, 1, \mathtt{s}, \mathtt{e}, \mathtt{m}$, M_R rechazará la entrada.

Así tenemos un candidato a ser codificación de transición en la quinta cinta de M_R. Recordemos que una transición válida es de la forma

$$\mathtt{e}q_{(2)}\mathtt{s}p_{(2)}\mathtt{e}q'_{(2)}\mathtt{s}p'_{(2)}\mathtt{m}\hat{\lambda}$$

donde $q, q' \in \{0, \ldots, s\}$, $p, p' \in \{1, \ldots, r\}$, $\hat{\lambda} \in \{0, 1\}$ y $q \neq 1, 2$. En este caso, M_R actúa en su quinta cinta de la siguiente manera:

1. M_R lee y borra e. En caso no encuentre el símbolo e, rechaza.

2. M_R escribe | al final de la cuarta cinta (inmediatamente después de $s_{(2)}$), y luego mueve uno por uno los caracteres de la quinta cinta hasta encontrar s; llamemos $q_{(2)}$ a la cadena de los caracteres desplazados. En caso encuentre un símbolo diferente de $0, 1, \mathtt{s}$, M_R rechaza. Si $q_{(2)}$ resulta ser 1 o 10, también rechaza.

3. M_R compara $q_{(2)}$ con $s_{(2)}$ (adaptando la máquina del ejemplo 3.7). Si $q > s$, M_R rechaza, en caso contrario, borra $q_{(2)}$ y | de la cuarta cinta.

4. M_R lee y borra s.

5. M_R escribe | al final de la tercera cinta (inmediatamente después de $n_{(2)}$), y luego mueve uno por uno los caracteres de la quinta cinta hasta encontrar e; llamemos $p_{(2)}$ a la cadena de los caracteres desplazados. En caso encuentre un símbolo diferente de 0, 1, e, M_R rechaza. Si $p_{(2)} = 0$, también rechaza.

6. M_R compara $p_{(2)}$ con $n_{(2)}$ (adaptando la máquina del ejemplo 3.7). Si $p > n$, M_R rechaza, en caso contrario, borra $p_{(2)}$ y | de la cuarta cinta.

7. M_R repite una vez más los pasos anteriores, con las siguientes diferencias: en el ítem 2, $q_{(2)}$ puede ser 1 o 10, y en el ítem 5, M_R moverá caracteres hasta encontrar el símbolo m.

8. Finalmente, M_R lee y borra m y verifica que el siguiente (y último) caracter sea 0 o 1. En caso de no serlo, o no ser el último, M_R rechaza.

En caso M_R no haya rechazado el contenido de la quinta cinta, deberá repetir el proceso anterior. M_R aceptará la entrada cuando la primera cinta esté vacía. □

3.2.2. La máquina de Turing universal

La **máquina de Turing universal**, que llamaremos U, es una máquina de Turing que, teniendo como entrada las codificaciones de una máquina M y una entrada w, acepta si y solamente si M acepta w.

Como $\langle M\rangle$ y $\langle w\rangle$ son palabras sobre C, una entrada válida para U será una palabra de la forma $\langle M\rangle\&\langle w\rangle$. Definimos el alfabeto de entrada de U como el conjunto $\Sigma_U = C \cup \{\&\}$. Así, U acepta $\langle M\rangle\&\langle w\rangle$ si, y solamente si, $w \in L_M$. Dicho de otra manera, el lenguaje de aceptación de U está dado por

$$L_U = \{\langle M\rangle\&\langle w\rangle \in \Sigma_U^* \ : \ M \text{ es máquina de Turing y } w \in L_M\}.$$

Definimos U como una máquina de Turing de cinco cintas con alfabeto de entrada Σ_U, y describiremos el funcionamiento de U. Para simplificar la descripción, supondremos que la entrada de U está en L_U, así, en caso U no pueda realizar alguna de las tareas que indicaremos, entonces U rechazará inmediatamente la entrada.

Inicialmente, U mueve $\langle w\rangle$ a su segunda cinta y luego decide si lo escrito en la primera cinta es una codificación válida. Ahora, suponiendo $\langle M\rangle = r_{(2)}|n_{(2)}|s_{(2)}\#\langle\delta\rangle$, U decide si $w \in \Sigma_M$, donde $\Sigma_M = \{1, \ldots, r\}$. Para comenzar la simulación, U copia $sn_{(2)}$ en la quinta cinta, borra la cabecera de $\langle M\rangle$ de la primera cinta, copia $\langle w\rangle$ y cambia el símbolo inicial s de $\langle w\rangle$ por $\dot{s}$ en la tercera cinta y escribe e0 en la cuarta cinta. Así, tenemos U como en la figura 3.1.

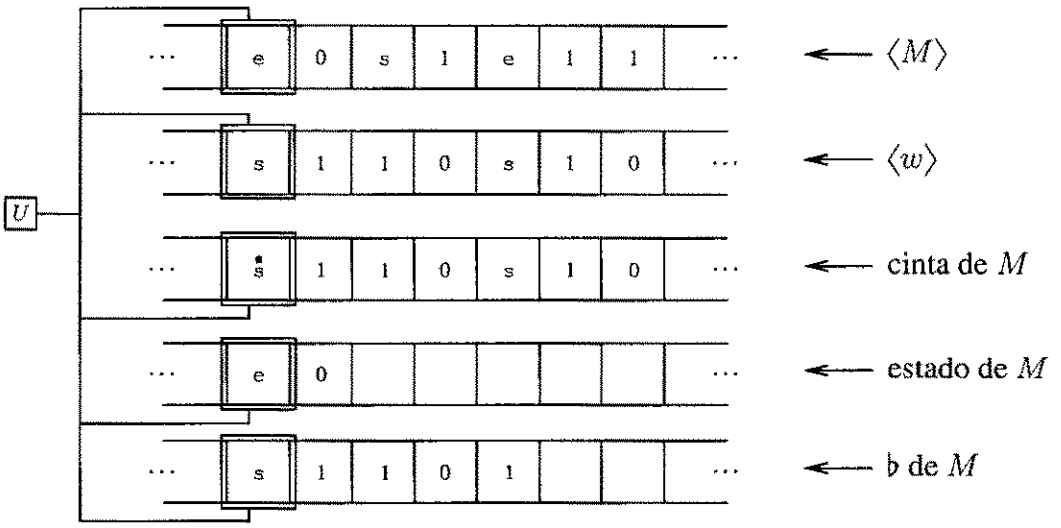


Figura 3.1: Máquina de Turing universal

Ahora comenzaremos la simulación de M con entrada w. En general, supongamos que el cabezal de la primera cinta apunta al caracter e que es el primer caracter de la codificación de una transición de la forma $\mathtt{e}q_{(2)}\mathtt{s}r_{(2)}\mathtt{e}q'_{(2)}\mathtt{s}r'_{(2)}\mathtt{m}\hat{\lambda}$. Supongamos también que tenemos un estado $\mathtt{e}q_{(2)}$ en la cuarta cinta y una cadena de la forma

$$\mathtt{s}p_{1(2)}\cdots\dot{\mathtt{s}}p_{i(2)}\cdots\mathtt{s}p_{t(2)}$$

en la tercera cinta, donde el cabezal de esta cinta apunta a $\dot{\mathtt{s}}$. En este caso $\dot{\mathtt{s}}$ denotará que el cabezal de M apunta a dicho caracter. Bajo estas condiciones U realiza lo siguiente:

1. Compara caracter a caracter la primera y cuarta cintas, mientras tales caracteres sean iguales. Cuando deje de ocurrir esto, si U encontró $(\mathtt{s},\mathtt{b})$ entonces pasa al siguiente paso; en caso contrario, el cabezal de la primera cinta pasa a la siguiente transición y el cabezal de la cuarta se ubica al principio del estado. En caso no haya siguiente transición, rechazar la entrada.

2. En este caso, el cabezal de la primera cinta apunta a s y el de la tercera apunta a $\dot{\mathtt{s}}$. Ahora U compara caracter a caracter la primera y tercera cintas, mientras tales caracteres sean iguales. Cuando deje de ocurrir esto, si U encuentra $(\mathtt{e},\mathtt{s})$ o $(\mathtt{e},\mathtt{b})$, pasa al siguiente paso; en caso contrario, el cabezal de la primera cinta pasa a la siguiente transición, el cabezal de la tercera se ubica nuevamente encima de $\dot{\mathtt{s}}$ y U pasa al ítem 1. En caso no haya siguiente transición, rechazar la entrada.

3. Este es el caso en que U encontró la transición de M que va a ejecutar. En este momento, el cabezal de la primera cinta apunta al segundo caracter e de la transición $\mathtt{e}q_{(2)}\mathtt{s}r_{(2)}\mathtt{e}q'_{(2)}\mathtt{s}r'_{(2)}\mathtt{m}\hat{\lambda}$. Luego, U borrará todo el contenido de la cuarta cinta, copiará caracter a caracter lo que encuentre en la primera cinta y dejará el cabezal de la cuarta cinta apuntando al siguiente caracter b a la derecha

de lo escrito. Observe que deberá haber copiado $eq'_{(2)}$ en la cuarta cinta. Después, U lee s y $\dot{s}$ en la primera y tercera cinta, respectivamente. Luego copia caracter a caracter la primera en la tercera cintas, hasta que: encuentre m en la primera cinta, o encuentre s o b en la tercera cinta. Pueden ocurrir cuatro posibilidades:

a) Si U encuentra m en la primera cinta y s o b en la tercera cinta, entonces pasamos al siguiente paso.

b) Si U encuentra m en la primera cinta y algún caracter diferente de s o b en la tercera cinta, entonces U borrará todo el contenido de la tercera cinta restante hasta encontrar s o b. Luego moverá todo el contenido de la tercera cinta desde dicho caracter hasta no dejar espacios vacíos.

c) Si U encuentra b en la tercera cinta, seguirá copiando caracteres de la primera cinta hasta encontrar m en esta.

d) Si U encuentra s en la tercera cinta, moverá todo el contenido a la derecha de s a la cuarta cinta y seguirá copiando caracteres de la primera cinta hasta encontrar m en esta. Luego, moverá nuevamente el contenido copiado en la cuarta cinta en su posición respectiva.

Para terminar la simulación de la ejecución de M, U leerá el caracter $\hat{\lambda}$ a la derecha de m y reemplazará $\dot{s}$ por s en la tercera cinta. Luego U buscará el siguiente s a la izquierda o derecha del s recién escrito y escribirá $\dot{s}$, en caso $\hat{\lambda}$ sea 0 o 1, respectivamente. En caso no haya tal s, U copiará $\dot{s}$ $n_{(2)}$ de la quinta cinta a la tercera, en la posición debida.

4. Finalmente, U verificará si lo escrito en la cuarta cinta es e1 o e10, y aceptará o rechazará la entrada, según sea el caso. De no suceder lo anterior, U posicionará los cabezales adecuadamente, para comenzar una nueva simulación.

Así, U aceptará la entrada $w' = \langle M \rangle \& \langle w \rangle$ si, y solamente si, encuentra e1 en su cuarta cinta, y esto ocurrirá si, y solamente si, M acepta w. En cualquier otro caso, U rechazará w'.

Hemos probado entonces la siguiente proposición.

Proposición 3.15. *El lenguaje* $L_U := \{\langle M \rangle \& \langle w \rangle : M \text{ es una máquina de Turing y } w \in L_M\}$ *es semidecidible.*

Además, modificando ligeramente el comportamiento de la máquina U, obtenemos:

Corolario 3.16. *El lenguaje* $\{\langle M \rangle \& \langle w \rangle : M \text{ es una máquina de Turing y rechaza } w\}$ *es semidecidible.*

3.2.3. No decidibilidad de lenguajes

El hecho de haber construido una máquina de Turing U que sólo reconoce L_U no demuestra que L_U *no* es decidible. En general, para probar que un lenguaje no es decidible, debemos mostrar que no

existe máquina de Turing que decida tal lenguaje. En esta sección, probaremos que existen lenguajes no decidibles, y daremos un ejemplo explícito.

Necesitaremos el siguiente lema, que resume las propiedades de los lenguajes decidibles y semidecidibles.

Lema 3.17. *Sea L un lenguaje sobre algún alfabeto Γ, y sea $\bar{L}$ el complemento de L, como en la sección 1.2.*

1. *Si L es decidible entonces es semidecidible.*
2. *Si L es decidible entonces $\bar{L}$ es decidible.*
3. *Si L y $\bar{L}$ son semidecidibles entonces L es decidible.*
4. *Si L no es decidible entonces, o bien L no es semidecidible, o bien $\bar{L}$ no lo es.*

Demostración:

1. Es inmediato de la definición.
2. Basta modificar la máquina de Turing que decide L para que acepte en lugar de rechazar y viceversa.
3. Sean M y M' las máquinas de Turing que reconocen L y $\bar{L}$, respectivamente. Construiremos una máquina N de dos cintas que decide L. Inicialmente N copia la entrada w en su segunda cinta y simulará M y M' en la primera y segunda cinta, respectivamente, y para esto, supondremos que $Q_M \times Q_{M'} \subset Q_N$. N comenzará la simulación en estado $(q_0^M, q_0^{M'})$ y terminará o bien cuando llegue a estado (q_a^M, q') o $(q, q_r^{M'})$, aceptando la entrada, o cuando llegue a estado $(q, q_a^{M'})$ o (q_r^M, q'), rechazando la entrada.
4. Es la afirmación contrarecíproca de 3. □

Gracias a lo hecho en la sección 3.2.1, obtenemos el siguiente resultado.

Proposición 3.18. *Existen lenguajes no semidecidibles.*

Demostración: La demostración es un simple argumento de conteo. Sea Σ un alfabeto finito; definamos el conjunto

$$\mathcal{L} = \{L \subset \Sigma^*\} = \mathcal{P}(\Sigma^*).$$

Tenemos que $\mathcal{L}$ es el conjunto de todos los lenguajes sobre Σ. Como Σ es finito entonces Σ^* es numerable y por un resultado bien conocido de análisis real, $\mathcal{L} = \mathcal{P}(\Sigma^*)$ tiene la misma cardinalidad que los números reales, es decir, es no numerable.

Por otro lado, definamos

$$S = \{L \subset \Sigma^* \ : \ L \text{ es un lenguaje semidecidible}\}.$$

Para cada $L \in S$, existe una máquina de Turing M que reconoce L, osea $L = L_M$. De esto, usando el axioma de elección, podemos construir una función inyectiva $\varphi : S \to \mathcal{L}_{TM}$ definida por $\varphi(L) = \langle M \rangle$, donde M es tal que $L = L_M$. Como $\mathcal{L}_{TM}$ es un lenguaje sobre el alfabeto finito $C = \{0, 1, \mathtt{s}, \mathtt{e}, \mathtt{m}, \#, |\}$, entonces $\mathcal{L}_{TM}$ es numerable y por lo tanto S lo es. Esto demuestra la proposición. □

Observación 3.19. La proposición anterior tiene una implicación filosófica importante: la cantidad de problemas que podemos plantear (reconocer lenguajes) es inmensamente mayor que la cantidad de problemas que podemos resolver (lenguajes semidecidibles). El hecho que las componentes (alfabeto, conjunto de estados, función de transición) de las máquinas de Turing sean finitos hace que sólo haya una cantidad numerable de lenguajes que podemos reconocer. Este fenómeno también ocurre en el caso de algoritmos.

¿Qué tipo de problemas son tan engorrosos que no pueden ser resueltos por máquinas de Turing? Mejor dicho, ¿qué tipo de lenguajes son tan complejos que no pueden ser reconocidos por máquinas de Turing? Por otro lado, aún habiendo lenguajes no semidecidibles, ¿habrá lenguajes semidecidibles pero no decidibles? La siguiente proposición responde positivamente a la anterior pregunta.

Proposición 3.20. *El lenguaje* $L_U = \{\langle M \rangle \& \langle w \rangle \ : \ M \text{ es máquina de Turing y } w \in L_M\}$ *no es decidible.*

Demostración: Supongamos, por reducción al absurdo, que exista una máquina de Turing H que decida L_U. En este caso, H acepta una entrada $z \in \Sigma_U^*$ si z es de la forma $\langle M \rangle \& \langle w \rangle$ con M máquina de Turing y $w \in L_M$, y rechaza z en cualquier otro caso.

Construiremos una máquina de Turing D con alfabeto de entrada Σ_U de la siguiente manera: con entrada W, D acepta W si, y solamente si, H rechaza $W\&W$. Como H decide L_U y $W\&W \in \Sigma_U^*$, entonces D se detiene para todo $W \in \Sigma_U^*$.

Analicemos el comportamiento de D con entrada $W_0 = \langle D \rangle$:

- Si D rechaza W_0 entonces H acepta la cadena $W_0\&W_0$. Esto implica que H acepta $\langle D \rangle \& W_0$, es decir, W_0 es aceptado por D, contradicción.
- Si D acepta W_0 entonces H rechaza la cadena $W_0\&W_0$. Sin embargo, como $W_0\&W_0 = \langle D \rangle \& W_0$ y $W_0 \in \Sigma_U^*$, entonces el hecho de que H rechaza $W_0\&W_0$ implica que D no acepta[2] W_0, nuevamente una contradicción.

[2] osea, o bien rechaza, o bien no se detiene.

Las contradicciones anteriores provienen de la construcción de D, que a su vez proviene de la existencia de H que decide L_U. Por lo tanto L_U no es decidible. □

Observación 3.21. En 1891, Georg Cantor publicó una demostración muy elegante de que los números reales no son numerables (cf. [Dehornoy, 2009, §1.5]). Este argumento es conocido como el "método de la diagonal" y consiste en enumerar los reales y construir un número real que no está en tal enumeración. En la demostración anterior hemos usado tal técnica para construir la máquina D a partir de la máquina H. En efecto, como $\mathcal{L}_{TM}$ es numerable, podemos fijar una enumeración $M_1, \ldots, M_n, \ldots$ de las máquinas de Turing. Como L_U es semidecidible, usando la máquina U podemos conocer si M acepta o rechaza $\langle N \rangle$ para algunos pares de máquinas M, N, es decir, tenemos la siguiente tabla incompleta:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1		rechaza	acepta		
M_2		acepta			
M_3	rechaza		acepta		
M_4	acepta		acepta	rechaza	
$\vdots$					$\ddots$

Al suponer la decidibilidad de L_U, suponemos la existencia de una máquina H que "complete" la tabla anterior. Así, tenemos la siguiente tabla (donde letra cursiva denota el resultado obtenido por H):

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*acepta*	rechaza	acepta	*acepta*	
M_2	*rechaza*	acepta	*rechaza*	*acepta*	
M_3	rechaza	*acepta*	acepta	*rechaza*	
M_4	acepta	*rechaza*	acepta	rechaza	
$\vdots$					$\ddots$

Para construir D tomamos la diagonal de la tabla anterior, y definimos D con entrada M como lo opuesto a la salida de H con entrada $\langle M \rangle \& \langle M \rangle$. La contradicción surge debido a que como D es también una

máquina de Turing, entonces $D = M_k$, para algún $k \in \mathbb{N}$. En este caso:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	***acepta***	rechaza	acepta	*acepta*			
M_2	*rechaza*	**acepta**	*rechaza*	*acepta*		$\vdots$	
M_3	rechaza	*acepta*	**acepta**	*rechaza*			
M_4	acepta	*rechaza*	acepta	**rechaza**			
$\vdots$					$\ddots$		
D	rechaza	rechaza	rechaza	acepta		¿?	
$\vdots$							$\ddots$

Así, obtenemos los siguientes corolarios.

Corolario 3.22. *El lenguaje* $\overline{L_U} = \Sigma_U^* \backslash L_U$ *no es semidecidible.*

Corolario 3.23. *El lenguaje* $L = \{\langle M \rangle \& \langle w \rangle \ : \ M$ *es máquina de Turing y rechaza* $w\}$ *no es decidible.*

Corolario 3.24. *El lenguaje* $L = \{\langle M \rangle \& \langle w \rangle \ : \ M$ *es máquina de Turing y se detiene con entrada* $w\}$ *es semidecidible pero no decidible.*

El corolario 3.24 hace una afirmación importante: no es posible decidir si una máquina M se detiene o no con entrada w, para toda posible máquina y para toda posible entrada. Este hecho se traduce en un problema práctico tangible en el caso de algoritmos: *el problema de la parada.*

3.3. Un problema no decidible

Nos interesa que restricciones tenemos que añadir a los algoritmos para que puedan ser ejecutados sin problemas por una computadora. Existen dos restricciones:

Restricciones aritméticas: Este problema surge cuando se desea representar números de precisión arbitraria en la computadora. Como la memoria disponible en una computadora es limitada, es imposible para las computadoras actuales representar cualquier número real. Este problema es estudiado por el análisis numérico.

Restricciones de ejecución: Este problema surge cuando se desea ejecutar una cantidad infinita de instrucciones. Por ejemplo considere las siguientes instrucciones:

```
Inicio
mientras 1 == 1 hacer
    escribir('hola mundo')
fin-mientras
Fin
```

Es claro que al ejecutarse las instrucciones anteriores el ejecutor no se detendrá nunca. Para darnos cuenta de esto, no hemos necesitado ejecutar las instrucciones anteriores, sino simplemente analizar las instrucciones.

La tarea de analizar un conjunto de instrucciones para determinar si termina o no, no es trivial la mayoría de las veces. Para darse cuenta de esto considere las siguientes instrucciones:

```
Inicio
perfecto := FALSO
n := 3
mientras (no perfecto) hacer
    s := 0
    d := 1
    mientras d < n hacer
        si n mod d == 0 entonces
            s := s + d
        fin-si
        d := d + 1
    fin-mientras
    si s == n entonces
        perfecto := VERDADERO
    sino
        n := n+2
    fin-si
fin-mientras
Fin
```

Al ejecutarse las instrucciones anteriores, estas se detendrán si, y sólo si, existe un número natural n tal que sea impar y perfecto (esto es, que n sea la suma de sus divisores propios). El hecho de que existan números perfectos impares es un problema abierto en teoría de números (cf. [Ribenboim, 2004, §2.VII,p. 83]). Luego, si se encuentra una manera de determinar si las instrucciones anteriores se detienen o no al ser ejecutadas, entonces habremos probado tal conjetura.

Es válido entonces preguntarse si existe alguna manera de determinar si un procedimiento efectivo es un algoritmo, es decir, si existe un algoritmo que diga si terminará la ejecución de un conjunto de instrucciones dado.

3.3.1. El problema de la parada

Conocido en inglés como *The Halting Problem*, este problema se pregunta lo siguiente:

"¿Existirá un algoritmo tal que: con entradas P y t, P cadena almacenando el pseudocódigo de un algoritmo y t almacenando una entrada de P; determine si el algoritmo P se detiene con entrada t, para todo algoritmo P y toda entrada t?"

Proposición 3.25. *No existe tal algoritmo.*

Demostración: Por reducción al absurdo, supongamos que si exista dicho algoritmo.

Algoritmo 3 (Halt).
Entrada: '(P, t)', con P cadena almacenando pseudocódigo de un algoritmo, y
t cadena almacenando una entrada para P.

```
Inicio
Sorprendente y maravilloso algoritmo
que resuelve el problema de la parada.
Fin
```

Salida: VERDADERO, si P se detiene con entrada t.

FALSO, si P denota el pseudocódigo de algún algoritmo y no se detiene con entrada t, o si P no denota el pseudocódigo de algún algoritmo.

Considerando el algoritmo anterior, construiremos un nuevo algoritmo.

Algoritmo 4 (Confunde).
Entrada: x cadena.

```
Inicio
si Halt(x,x) == VERDADERO entonces
    mientras 1 == 1 hacer
        escribir('hola mundo')
    fin-mientras
sino
    y := VERDADERO
fin-si
Fin
```

Salida: y

Sea C una cadena almacenando el pseudocódigo anterior y analicemos el resultado de `Halt` (C, C). Existen dos posibilidades para la respuesta:

- `Halt` (C, C) es VERDADERO. Esto significa que `Confunde` con entrada C se detiene (pues C denota el código de `Confunde`). Esto significa que al evaluarse la sentencia `si` de `Confunde`, la

expresión `Halt(x,x)==VERDADERO` resultó falsa para `x=`C, pues en caso contrario se hubiera ejecutado un bucle infinito. Luego `Halt(C,C)` es falso, lo que implica que C no se detiene con entrada C. Esto es una contradicción.

- `Halt(`C`,`C`)` es `FALSO`. Esto significa que `Confunde` no se detiene con entrada C. De acuerdo a la definición de `Confunde` esto puede darse, o bien porque `Halt(x,x)` no se detuvo al ejecutarse o bien `Halt(x,x)==VERDADERO` resultó verdadera para `x=`C. Lo primero no puede darse porque estamos asumiendo que `Halt` es un algoritmo (osea, se detiene con cualquier entrada). Sin embargo, lo segundo tampoco puede darse, pues si `Halt(x,x)==VERDADERO` resultó verdadera para `x=`C, entonces `Halt(`C`,`C`)` es `VERDADERO`, lo cual es una contradicción.

Como en ambos casos se llegó a una contradicción, tenemos que no es posible la existencia del algoritmo `Halt`. □

Como ya mencionamos, las proposiciones 3.25 y 3.18 tienen una seria consecuencia: existen problemas [resp. lenguajes] para los cuales no es posible encontrar un algoritmo [resp. máquina de Turing] que lo resuelva [resp. decida]. Existe un análogo de este comportamiento en el caso de matemáticas. Esta patología tiene un análogo en matemáticas. En 1931, Kurt Gödel demostró que en cualquier sistema axiomático suficientemente "grande" existe afirmaciones verdaderas que no son demostrables (cf. [Gödel, 1931]).

Capítulo 4

Conclusiones

La computación teórica nació a partir de la teoría de la computabilidad, y esta nació a partir del décimo problema de Hilbert y de los trabajos de Church, Gödel, Turing, etc. El primer objetivo de este trabajo fue el de estudiar los modelos de computabilidad de los autores antes mencionados. Estos modelos fueron hechos en base a las funciones con dominio y rango en los números naturales y sirven para establecer la noción de función computable, esto es, aquellas funciones cuya regla de correspondencia puede ser calculada, en cualquier punto, por un computador mediante un algoritmo.

Primeramente, presentamos el lambda cálculo de Church, comenzando con la noción de λ-término. Por si solo, un λ-término denota evaluaciones y/o composiciones de una o más funciones. Luego establecimos reglas para evaluar y reducir λ-términos. Con esto, pudimos modelar las nociones de número natural y de funciones naturales. Aún mas, también vimos que, usando λ-términos, es posible definir las nociones de "verdadero" y "falso", y modelar así el cálculo proposicional.

A continuación, presentamos a las funciones recursivas primitivas. Estas consisten de componer, mediante las operaciones de substitución y recursión primitiva, ciertas funciones básicas, que llamamos "funciones iniciales". No es difícil ver que una gran cantidad de funciones naturales son recursivas primitivas. Sin embargo, un ejemplo de función no recursiva primitiva es la conocida función de Ackermann. Probamos que la función de Ackermann es superior a cualquier función recursiva primitiva, por ende, esta no es recursiva primitiva. Podemos capturar la función de Ackermann definiendo la noción de función recursiva. Para esto, definimos la operación de minimización, que esencialmente busca el menor número natural que anula una función. Usando el operador de minimización, junto con las operaciones de substitución y recursión primitiva, y las funciones iniciales, pudimos definir una familia de funciones naturales, no todas totalmente definidas, que llamamos "funciones recursivas parciales". Así, las funciones recursivas son las funciones recursivas parciales que son totalmente definidas, y es posible probar que la función de Ackermann es recursiva.

Finalmente, presentamos a las máquinas de Turing. Una máquina de Turing es un cabezal de funcionamiento autónomo que actúa sobre una cinta infinita. En este caso, la cinta es considerada como un arreglo unidimensional infinito, donde cada "casilla" de la cinta contiene un símbolo, y el cabezal en

cada paso modifica el contenido de la casilla a la que apunta y se mueve una casilla a la derecha o a la izquierda. Podemos considerar una máquina de Turing como una función, que dado un contenido inicial de las casillas de la cinta, devuelve el contenido de la cinta que se obtiene cuando el cabezal se detiene. Para convencernos del poder computacional que tiene una máquina de Turing, definimos un modelo de máquina que admite múltiples cintas y probamos que este modelo es equivalente al modelo original de una cinta.

Los tres modelos de computabilidad presentados son equivalentes. Esto llevó a Alonzo Church a afirmar que cualquiera de estos modelos sirve como formalización de la noción de algoritmo.

La primera parte de este trabajo termina mostrando un ejemplo de función no computable, esto es, una función natural, definida en todo $\mathbb{N}$, tal que no existe algoritmo que pueda calcularla. Esta función es conocida como la función del *castor ocupado*.

El segundo objetivo de este trabajo fue estudiar el concepto de decidibilidad. Los problemas de decidibilidad son aquellos cuya respuesta es "si" o "no", y para esto, usamos a las máquinas de Turing para formalizar el concepto de algoritmo, debido a su versatilidad. Para el estudio de este tipo de problemas, modificamos la definición de máquina de Turing de tal manera que distinga como respuesta dos estados finales específicos: estado de aceptación y estado de rechazo. Así, un problema de decisión es modelado por un problema de pertenencia de un elemento (que será la entrada de la máquina) en un lenguaje (que determinará la definición de la máquina), y una solución de un problema de decisión es una máquina de Turing tal que determine la pertenencia al lenguaje de su entrada. En este caso, un lenguaje se dice *decidible* si problema de decisión que define tiene solución.

De la misma manera que existen funciones no computables, es posible definir, en el contexto de máquinas de Turing, lenguajes *no decidibles*. Para esto, construimos la *máquina de Turing universal*, que esencialmente es una máquina que ejecuta otras máquinas en entradas dadas. De manera similar, ya en el contexto de algoritmos, definimos el *problema de la parada*, el cual es también no decidible. Similarmente al caso de máquinas de Turing, probamos que no existe algoritmo que resuelva el problema de la parada.

Los temas expuestos en este trabajo son los principios de la teoría de la computabilidad, y la continuación natural del trabajo es la teoría de la complejidad. Esta teoría estudia la cantidad de pasos que un algoritmo usa para resolver un problema,y para su estudio, la noción de *no determinismo* es especialmente útil. De aquí surge el problema principal de la teoría de la computación: el problema $\mathcal{P}$ vs $\mathcal{NP}$.

Apéndice A

Citas

A.1. El décimo problema de Hilbert

10. Entscheidung der Lösbarkeit einer diophantischen Gleichung.

Eine diophantische Gleichung mit irgendwelchen Unbekannten und mit ganzen rationalen Zahlkoefficienten sei vorgelegt: *man soll ein Verfahren angeben, nach welchen sich mittels einer endlichen Anzahl von Operationen entscheiden läßt, ob die Gleichung in ganzen rationalen Zahlen lösbar ist.*

A.2. Definición original de la máquina de Turing

El siguiente texto fue extraído literalmente de *"On computable numbers, with an application to the Entscheidungsproblem"*, publicado por Alan Turing (cf. [Turing, 1936]).

1. Computing machines.

We have said that the computable numbers are those whose decimals are calculable by finite means. This requires rather more explicit definition. No real attempt will be made to justify the definitions given until we reach §9. For the present I shall only say that the justification lies in the fact that the human memory is necessarily limited.

We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions $q_1, q_2, \ldots, q_R$ which will be called "m-configurations". The machine is supplied with a "tape" (the analogue of paper) running through it, and divided into sections (called "squares") each capable of bearing a "symbol". At any moment there is just one square, say the r-th, bearing the symbol $\mathfrak{S}(r)$ which is "in the machine". We may call this square the "scanned square". The symbol on the scanned square may be called the "scanned symbol". The "scanned symbol" is the only one of which the machine is, so to speak, "directly

aware". However, by altering its m-configuration the machine can effectively remember some of the symbols which it has "seen" (scanned) previously. The possible behaviour of the machine at any moment is determined by the m-configuration q_n and the scanned symbol $\mathfrak{S}(r)$. This pair $q_n, \mathfrak{S}(r)$ will be called the "configuration": thus the configuration determines the possible behaviour of the machine. In some of the configurations in which the scanned square is blank (i.e. bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations the m-configuration may be changed. Some of the symbols written down will form the sequence of figures which is the decimal of the real number which is being computed. The others are just rough notes to "assist the memory". It will only be these rough notes which will be liable to erasure.

It is my contention that these operations include all those which are used in the computation of a number. The defence of this contention will be easier when the theory of the machines is familiar to the reader. In the next section I therefore proceed with the development of the theory and assume that it is understood what is meant by "machine", "tape", "scanned", etc.

A.3. Church acerca de la definición de efectivamente computable

El siguiente texto fue extraído literalmente de *"An Unsolvable Problem of Elementary Number Theory"*, publicado por Alonzo Church (cf. [Church, 1936]).

As will appear, this definition of effective calculability can be stated in either of two equivalent forms, (1) that a function of positive integers shall be called effectively calculable if it is λ-definable in the sense of §2 below, (2) that a function of positive integers shall be called effectively calculable if it is recursive in the sense of §4 below. The notion of λ-definability is due jointly to the present author and S. C. Kleene, successive steps towars it having been taken by the presen author in the *Annals of Mathematics*, vol. 34 (1933), p. 863, and by Kleene in the *American Journal of Mathematics*, vol. 57 (1935), p. 219. The notion of recursiveness in the sense of §4 below is due jointly to Jacques Herbrand and Kurt Gödel, as is there explained. And the proof of equivalente of the two notions is due chiefly to Kleene, but also partly to the present author and to J. B. Rosser, as explained below. The proposal to identify these notions with the intuitive notion of effective calculability is first made in the present paper (but see the first footnote to §7 below).

With the aid of the methods of Kleene (*American Journal of Mathematics*, 1935), the considerations of the present paper could, with comparatively slight modification, be carried through entirely in terms of λ-decidability, without making use of the notion of recursiveness. On the other hand, since the results of the present paper were obtained, it has been shown by Kleene (see his forthcoming paper, "General recursive functions of natural numbers") that analogous re-

sults can be obtained entirely in terms of recursiveness, without making use of λ-definability. The fact, however, that two such widely different and (in the opinion of the author) equally natural definitions of effective calculability turn out to be equivalent adds to the strength of the reasons adduced below for believing that they constitute as general a characterization of this notion as is consistent with the usual intuitive understanding of it.

Bibliografía

[Church, 1932] Church, A. (1932). A set of postulates for the foundation of logic. *Ann. of Math. (2)*, 33(2):346–366.

[Church, 1933] Church, A. (1933). A set of postulates for the foundation of logic. *The Annals of Mathematics*, 34(4):839–864.

[Church, 1936] Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363.

[Church and Rosser, 1936] Church, A. and Rosser, J. B. (1936). Some properties of conversion. *Trans. Amer. Math. Soc.*, 39(3):472–482.

[Cook, 2006] Cook, S. (2006). The P versus NP problem. In *The millennium prize problems*, pages 87–104. Clay Math. Inst., Cambridge, MA.

[Curry, 1934] Curry, H. B. (1934). Some properties of equality and implication in combinatory logic. *Ann. of Math. (2)*, 35(4):849–860.

[Cutland, 1980] Cutland, N. (1980). *Computability*. Cambridge University Press, Cambridge. An introduction to recursive function theory.

[Davis et al., 1961] Davis, M., Putnam, H., and Robinson, J. (1961). The decision problem for exponential diophantine equations. *Ann. of Math. (2)*, 74:425–436.

[Dehornoy, 2009] Dehornoy, P. (2009). Cantor et les infinis. *Gazette des Mathèmaticiens*, (121):29–46.

[Gödel, 1931] Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatsh. Math. Phys.*, 38(1):173–198.

[Hermes, 1965] Hermes, H. (1965). *Enumerability, decidability, computability. An introduction to the theory of recursive functions*. Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete, Band 127. Translated by G. T. Herman and O. Plassmann. Academic Press Inc., New York.

[Hopcroft et al., 2001] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2nd edition.

[Kleene, 1935] Kleene, S. C. (1935). A theory of positive integers in formal logic. part ii. *American Journal of Mathematics*, 57(2):219–244.

[Kleene, 1936] Kleene, S. C. (1936). λ-definability and recursiveness. *Duke Math. J.*, 2(2):340–353.

[Kleene and Rosser, 1935] Kleene, S. C. and Rosser, J. B. (1935). The inconsistency of certain formal logics. *Ann. of Math. (2)*, 36(3):630–636.

[Lewis and Papadimitriou, 1997] Lewis, H. R. and Papadimitriou, C. H. (1997). *Elements of the Theory of Computation*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Lucchesi et al., 1979] Lucchesi, C. L., Simon, I., Simon, I., Simon, J., and Kowaltowski, T. (1979). *Aspectos teóricos da computação*, volume 8 of *Projeto Euclides [Euclid Project]*. Instituto de Matemática Pura e Aplicada, Rio de Janeiro.

[Matijasevič, 1970] Matijasevič, J. V. (1970). The Diophantineness of enumerable sets. *Dokl. Akad. Nauk SSSR*, 191:279–282.

[Radó, 1962] Radó, T. (1962). On non-computable functions. *Bell System Tech. J.*, 41:877–884.

[Ribenboim, 2004] Ribenboim, P. (2004). *The little book of bigger primes*. Springer-Verlag, New York, second edition.

[Robinson, 1969] Robinson, J. (1969). Unsolvable diophantine problems. *Proc. Amer. Math. Soc.*, 22:534–538.

[Sipser, 1996] Sipser, M. (1996). *Introduction to the Theory of Computation*. PWS Pub Co, Boston, MA.

[Tenenbaum and Augenstein, 1983] Tenenbaum, A. M. and Augenstein, M. J. (1983). *Estructura de Datos en Pascal*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Turing, 1936] Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265.