UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO

FACULTAD DE CIENCIAS FÍSICO MATEMÁTICAS "MAT. LUIS MANUEL RIVERA GUTIERREZ"

EQUIVALENCIA DE MODELOS COMPUTACIONALES

Tesis para optar al grado de licenciado en Ciencias Físico Matemáticas.

Autor:

Gerardo Lauro Maldonado Martínez

Supervisado por:

Dra. Karina Mariela Figueroa Mora

Morelia, Michoacán
abril, 2016

Este trabajo lo dedico a mi familia que me ha apoyado en lo que se les ha hecho posible. También agradezco a Karina Figueroa que siempre ha servido como una guía y en especial una amiga.

Resumen

Hoy en día las ciencias de la computación han tomado una importancia fundamental por si mismas, en otras ramas de las ciencias y en la vida cotidiana. En este trabajo se revisa un resultado que forma parte de los testigos de la Tesis de Church-Turing, enunciado que en síntesis afirma que todo modelo de computación construible tiene menor ó el mismo poder de cómputo que una Máquina de Turing.

En primer lugar se repasan algunos conceptos básicos sobre conjuntos, alfabetos y funciones, así como la definición y otras cosas a tener en cuenta sobre las Máquinas de Turing. Siguiendo se enunciará la Tesis de Church-Turing haciendo énfasis en algunos aspectos de esta que servirán como preámbulo de los otros 2 modelos computacionales que son tratados.

Después seguiremos con una sección sobre una forma de describir Máquinas de Turing más fácil, así como la descripción de la Máquina Ábaco y las funciones recursivas. Luego aprovechando de la facilidad de probar que funciones aritméticas son recursivas se revisa que estas funciones pueden ser computadas por Ábacos y Máquinas de Turing.

Para finalizar se probó un lema (y algunas otras proposiciones) que nos permiten solo considerar funciones de números naturales y 3 símbolos para las máquinas de Turing lo cual facilita la prueba de que las funciones computables por estos modelos son exactamente las funciones recursivas.

Palabras clave: Computación, Matemáticas, Lógica, Turing, Recursivo.

Abstract

Today computer science has taken fundamental importance itself and in another areas of science. In this paper we will review a esencial result for "working hypothesis" known as Church-Turing thesis, sentence which states that all computational models have minor or the same power as Turing machine.

In first place we overview some basics about sets, alphabets and functions, as well as the definition and another things to keep in mind about Turing Machines. Following this way, we introduce the Church-Turing Thesis.

After that, we have a section about another form to represent Turing Machines and we also introduce two models of computation: The Abacus Machine and The Recursive Functions. Next to take advantage of fact that recursive functions are easy for arithmetic, we will show a proof that every recursive function is Turing-computable across the Abacus Machine, with the purpose of have this arithmetic functions in our catalog of Turing-computable functions.

On the final we enumerate the Turing machines, prove another lemas and close this with the fact that a function is Turing-computable if only if is Abacus-computable if only if is recursive.

Índice general

Capítulo 1

Introducción

Hoy en día el modelo computacional más conocido y utilizado es la máquina RAM (*Random Access Machine*), sin embargo este es fundamentalmente equivalente a cualquier modelo computacional construible.

En este trabajo comenzaremos mostrando el más popular de los modelos: la Máquina de Turing, después se expondrá otro modelo aparentemente menos poderoso, un mecanismo bastante básico conocido como Máquina Ábaco. Posteriormente se presentarán las funciones recursivas, que son de hecho justamente las funciones computables por estos modelos y por último la equivalencia entre ambos modelos y tales funciones.

1.1. Conceptos preliminares

Para abordar los resultados que se presentarán, es necesario contar con la noción de ciertos conceptos que se repasarán en esta sección iniciando con el concepto de número natural.

Básicamente haremos uso de los naturales justo como se presentan en la teoría de conjuntos con los axiomas ZF (Zermelo-Fraenkel) [1], es decir describiéndolos de la siguiente manera:

- $0 = \emptyset$.
- $s(n) = n \cup \{n\}$.

Donde $s : \mathbb{N} \to \mathbb{N}$ es la función que devuelve el sucesor del argumento. De este modo la construcción queda definida de la siguiente manera:

$$0 = \emptyset$$

$$1 = s(0) = 0 \cup \{0\} = \{0\}$$

$$2 = s(1) = 1 \cup \{1\} = \{0, 1\}$$

$$\vdots$$

$$s(n) = \{0, 1, 2, ..., n\}$$

Dicho esto continuaremos con el concepto de cardinalidad que extiende el concepto de tamaño que se tiene de los conjuntos finitos.

Definición 1.1.1. *Se dice que dos conjuntos A y B tienen la misma cardinalidad si existe una función $f : A \to B$ biyectiva y se denota por $|A| = |B|$. Además se dice que A tiene cardinalidad menor o igual que B si existe una función $f : A \to B$ inyectiva y lo denotamos por $|A| \leq |B|$.*

De la definición es claro que si $A = \{a_1, ..., a_n\}$ entonces $|A| = |n|$ y el conjunto vacío tiene la misma cardinalidad que el 0 (son el mismo conjunto). También queda claro que los naturales están ordenados por tamaño de la misma manera que por cardinalidad, esto muestra que en efecto la definición es una extensión del concepto de tamaño finito.

Definición 1.1.2. *Sea A un conjunto entonces se define el conjunto potencia $P(A)$ de A como sigue:*

$$P(A) = \{B \mid B \subseteq A\}$$

El conjunto potencia de algún conjunto, es el conjunto de todos sus subconjuntos.

El siguiente teorema muestra (contrario a la intuición común) la existencia de mas de un tipo de infinito.

Teorema 1.1.3. *Sea A un conjunto, entonces no existe función suprayectiva de A en $P(A)$.*

Demostración. Supongamos que existe $f : A \to P(A)$ suprayectiva, entonces consideremos.

$$V = \{x \in A \mid x \notin f(x)\}$$

Como f es suprayectiva existe $y \in A$ tal que $f(y) = V$.
Si $y \in V$, entonces $y \notin f(y)$ y por tanto $y \notin V$!.
Si $y \notin V$, entonces $y \in f(y)$ y por tanto $y \in V$!.
La contradicción viene de la suposición de la existencia de f, por tanto f no existe. □

Corolario 1.1.4. $|\mathbb{N}| < |P(\mathbb{N})|$.

A continuación se muestran algunos ejemplos de conjuntos que si tienen la misma cardinalidad que los naturales aunque se tenga la contención estricta de uno en el otro, lo cual no ocurre en conjuntos finitos.

Ejemplo 1.1.1. $|\mathbb{N}| = |\mathbb{Z}|$ *con la función* $f : \mathbb{N} \to \mathbb{Z}$ *tal que:*

$$f(n) = (-1)^n \left\lfloor \frac{n}{2} \right\rfloor$$

Donde $\lfloor\rfloor : \mathbb{Q} \to \mathbb{N}$ *es la función piso, tal que* $\forall q \in \mathbb{Q}$ $\lfloor q \rfloor$ *es el natural mas cercano, menor o igual a* q*.*

Ejemplo 1.1.2. $|\mathbb{N}| = |2\mathbb{N}|$ *con la función* $f : \mathbb{N} \to 2\mathbb{N}$ *tal que:*

$$f(n) = 2n$$

Donde $2\mathbb{N}$ *es el conjunto de números (naturales) pares.*

Definición 1.1.5. *Se dice que un conjuntos* A *es numerable si* $|A| = |\mathbb{N}|$*.*

Por otro lado es necesario revisar el concepto de función parcial, básicamente una función parcial es una relación que cumple que a cada elemento del dominio le toca uno o ningún elemento del codominio a diferencia de una función (total) donde a cada elemento del dominio le toca un elemento del codominio.

Definición 1.1.6. *Sean* A *y* B *conjuntos, decimos que una relación* $R \subset A \times B$ *es función parcial si* $\forall a \in A$ *se cumple alguna de las siguientes 2 condiciones.*

- $\exists! b \in B$ *tal que* $(a, b) \in R$
- $\forall b \in B$ $(a, b) \notin R$

Ejemplo 1.1.3. *Sea* $f : \mathbb{N} \to \mathbb{N}$ *tal que*

$$f(n) = \begin{cases} n & \text{si } n \text{ es par} \\ indefinida & \text{si } n \text{ es impar} \end{cases}$$

Además f *con su dominio restringido a los pares es una función total.*

La definición que sigue dice cuando un conjunto es enumerable. Básicamente un conjunto A es enumerable si se pueden numerar sus elementos, es decir si se pueden representar sus elementos como $\{a_0, a_1, a_2, ...\}$.

Definición 1.1.7. *Sea* A *un conjunto y* $f : \mathbb{N} \to A$ *una función parcial o total. Decimos que* f *es una enumeración de* A *si* f *es suprayectiva. En tal caso* A *es enumerable.*

Hay que notar que en la enumeración de un conjunto varios naturales pueden tener por imagen al mismo elemento, pero solo consideraremos enumeraciones inyectivas y continuas en el sentido de que comienzan en el 0 sin saltos, esto a menos que se especifique lo contrario.

Ejemplo 1.1.4. *El conjunto de números primos* $\mathbb{P}$ *es enumerable, entonces es posible referirse a los primos como* $\{p_0, p_1, p_2, ...\}$*.*

De ahora en adelante cuando me refiera a una función puede ser parcial o total, para referirnos a solo un tipo se hará explícitamente.

Definición 1.1.8. *Sea A un conjunto entonces se define:*

$$A^{\mathbb{N}} = \{\ f : \mathbb{N} \to A \mid f \text{ es función total } \}$$

La definición anterior de producto numerable permite representar a los elementos como tuplas con infinitas entradas pero puede generar ambigüedad por lo que se opta por definirlo con funciones.

Para el análisis de sistemas lógicos una técnica muy útil es fijarse no en el contenido de las proposiciones sino en el lenguaje en el que están escritas, lo mismo ocurre en la computación donde en realidad todo lo escrito, por ejemplo este texto, es convertido a números mediante una serie de reglas explícitas para su computación e interpretación por la computadora.

Definición 1.1.9. *Se denomina alfabeto a cualquier conjunto finito cuyos elementos se nombran caracteres. Las sucesiones finitas de caracteres se denominan cadenas.*

Definición 1.1.10. *Dado un alfabeto Σ el conjunto de todas las cadenas posibles que se pueden formar con sus caracteres se denotara como Σ^*. Así como $\forall n \in \mathbb{N}$ el símbolo Σ^n denota el conjunto de todas las cadenas de largo a lo más n.*

Observación. $\Sigma^* = \bigcup_{i=1}^{\infty} \Sigma^i$

Ejemplo 1.1.5. *Sea $\Sigma = \{a, b\}$ un alfabeto y algunas cadenas que podemos formar con este son: aa, a, baba, abab, abba, abaaaa, ϵ. Donde ϵ denota la cadena vacía. La cadena vacía en efecto es una cadena, es la sucesión de caracteres finita de 0 elementos.*

Con los conceptos de alfabeto y cadena definidos formalmente es posible hacer proposiciones sobre estos, hay que notar que un lenguaje sobre un alfabeto Σ tiene que ser un subconjunto de Σ^* generado por algunas reglas, justo como el español, el inglés o cualquier lengua.

Sobre cadenas podemos definir varias operaciones que son de utilidad como la concatenación $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$ que dadas 2 cadenas $w, v \in \Sigma^*$ nos devuelve otra cadena con sus caracteres pegados, es decir si $w = w_1 w_2 ... w_n$ y $v = v_1 v_2 ... v_m$ entonces:

$$w \cdot v = w_1 ... w_n v_1 ... v_m$$

Otra operación de la que se hará uso mas tarde es $first : \Sigma^* \setminus \{\epsilon\} \to \Sigma$ que devuelve el primer caracter de la cadena dada.

Antes de continuar hay que tener en cuenta las funciones proyección p_i que dada una tupla, devuelven el contenido de la i-esima coordenada. Por ejemplo $first$ puede verse como la proyección en la primera coordenada.

Un concepto que también será de utilidad es el de sucesión casi finita que se define a continuación.

Definición 1.1.11. *Se dice que una sucesión $f \in \mathbb{N}^{\mathbb{N}}$ es casi finita si existe $N \in \mathbb{N}$ tal que $\forall N \leq n$ se tiene $f(n) = 0$.*

Ejemplo 1.1.6. *Las cadenas de caracteres sobre un alfabeto Σ se pueden ver como sucesiones casi finitas considerando una enumeración e de Σ, así la cadena abbba se puede ver como:*

$$(e^{-1}(a), e^{-1}(b), e^{-1}(b), e^{-1}(b), e^{-1}(a), 0, 0, ...)$$

1.2. Máquina de Turing

Una vez revisados los conceptos preliminares presentaré el modelo de computo más conocido, propuesto por Alan Mathison Turing (1936).

Una Máquina de Turing (abreviada MT desde ahora) consta de una cinta dividida en casillas de largo infinito además de un cabezal que puede moverse a lo largo de ésta (véase Figura 1.1), sobrescribir el contenido de una casilla y leer el contenido de alguna otra. Cada casilla solo puede contener un carácter del alfabeto predefinido que usara la MT.

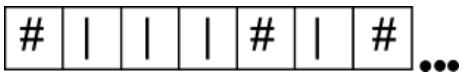

Figura 1.1: Ejemplo de cinta de una Máquina de Turing.

Con el concepto intuitivo revisado, es momento de dar una definición para la Máquina de Turing.

Definición 1.2.1. *Sean $X \subset Y$ alfabetos con $\# \in Y \backslash X$ un carácter distinguido. Una MT M con alfabeto de cinta X es una tupla:*

$$M = \{Y, Q, \delta, q_i, q_f\}$$

tal que:

- *Q es un conjunto finito cuyos elementos llamaremos estados.*
- *δ es una función parcial*

  $$\delta : Q \times Y \to Q \times Y \times \{\triangleright, \triangleleft, \Lambda\}$$

  Llamada función de transición. $\delta(q_f, a)$ no debe estar definida para ningún caracter de Y.
- *q_i : es el estado inicial de M.*
- *q_f : es el estado final de M.*

La definición puede no decir mucho en primera instancia, por lo que a continuación se explican más a detalle los elementos que componen esta maquinaria.

Primero hay que notar que en X se encuentran los posibles caracteres que pueden estar escritos en la cinta antes de encender la máquina, a diferencia de Y que son los caracteres que la máquina puede reconocer y escribir en la cinta. El símbolo # tiene la tarea de ser el espacio en blanco (para separar cadenas, justo como en la vida cotidiana) y es el único caracter fuera de X que puede aparecer en la cinta de entrada.

Por otro lado Q y δ son el código en el que está escrita la máquina, dado el estado actual y el carácter bajo el cabezal δ determina a que estado pasar, que carácter escribir (puede escribir el mismo, para simular la acción de no escribir nada) y mover el cabezal a la derecha ($\triangleright$), izquierda ($\triangleleft$) o dejarlo en el mismo lugar (Λ).

El estado en el que comienza la máquina es q_i y si la máquina pasa al estado q_f se entiende que la máquina se detuvo. Antes de exponer ejemplos y continuar con las máquinas es conveniente fijar la notación que se usará para representar números naturales que en realidad es bastante simple, usando un | para representar el 0, || para el 1 y así sucesivamente.

Ejemplo 1.2.1. *Máquina sucesor (dado $n \in \mathbb{N}$ devuelve $s(n)$).*

Entrada. *La cinta de entrada tendrá en la primera casilla un # seguida de una sucesión de | cuya cantidad representa un número natural y finaliza con una sucesión infinita de símbolos # (ver Figura 1.2).*

Figura 1.2: Ejemplo de entrada para la MT sucesor.

Salida. *La cinta contará con una configuración similar a la de entrada pero con un | mas al final de la sucesión que representa el número (ver Figura 1.3).*

Figura 1.3: Ejemplo de salida para la MT sucesor.

Cabe observar que hay que describir δ, Q y los alfabetos, pero al describir la función parcial quedaran descritos los estados. En este ejemplo $X = \{|\}$ y $Y = \{|, \#\}$.

En el Cuadro 1.1 queda descrita toda la máquina la cual realiza la tarea de dado $n \in \mathbb{N}$ (en nuestra codificación de | y #) nos devuelve $s(n)$ (donde s es la función sucesor en los naturales). Cabe resaltar que $q_i = q_0$ en este caso.

El siguiente paso, es mostrar una manera formal de definir cuando una máquina termina un proceso correctamente (es decir dada una entrada, deja al detenerse en la cinta el resultado esperado), concentrándonos en que tipo de

q	a	$\delta(q,a)$
q_0	#	$(q_1, a, \triangleright)$
q_1	#	$(q_f, \|, \Lambda)$
q_1	|	$(q_1, a, \triangleright)$

Cuadro 1.1: Descripción de la función de transición

funciones puede computar, es decir, dado un elemento del dominio, escriba en la cinta su imagen.

Definición 1.2.2. *Una configuración de una MT es un elemento de* $C_M = Y^* \times Q \times Y^* \cdot (Y \setminus \{\#\})$.

En la definición anterior $Y^* \cdot (Y \setminus \{\#\})$ se refiere a cadenas de Y^* concatenadas con un caracter diferente a # al final.

Las configuraciones denotan como se encuentra la MT en cierto momento y estas son útiles para analizar que tipo de procedimientos se pueden hacer con la máquina. Por convención se toma que en una configuración (u, q, v) el caracter $first(v)$ es el que se encuentra debajo del cabezal. Es visible que antes de arrancar una MT su configuración es (ϵ, q_i, v) con $v \in (X \cup \{\#\})^*$. Para evitar denotar explicítamente la sucesión infinita de símbolos # se toma en cuenta que después de v se encuentra dicha sucesión.

Definición 1.2.3. *Una configuración* (u, q, v) *es admisible si* $u \neq \epsilon$ *ó* $u = \epsilon$ *y* $p_3(\delta(q, first(v))) \neq \triangleleft$. *Aquí* p_3 *es la proyección en la tercera coordenada.*

La siguiente definición nos provee una forma para analizar si dado un estado, M puede llegar a otro mediante sus operaciones definidas.

Definición 1.2.4. *Sea* M *una MT y* $c_1 = (u, q, v)$, $c_2 = (u', q', v')$ *configuraciones admisibles de* M, *se define la relación* $\vdash_M$ *"lleva en un paso" en* $C_M \times C_M$ *donde* $c_1 \vdash_M c_2$ *si* $\delta(q, first(v))$ *deja la máquina en la configuración* c_2.

La relación $\vdash_M^n$ *"lleva en n pasos" se define como* $c1 \vdash_M^n c_n$ *sí y solo sí existe una sucesión* $c_1 \vdash_M ... \vdash_M c_n$.

La relación $\vdash_M^*$ *"lleva en cero ó más pasos" se define como* $c_1 \vdash_M^* c_2$ *sí y solo sí existe* $n \in \mathbb{N}$ *tal que* $c_1 \vdash_M^n c_2$.

Cuando no cause ambigüedad se omitirá la máquina M en la notación de estas relaciones.

Definición 1.2.5. *Sea* M *una MT, una computación de* M *es una sucesión* $c_1 \vdash_M ... \vdash_M c_n$ *de configuraciones admisibles tal que:*

- $c_1 = (\epsilon, q_i, v)$
- $c_n = (\epsilon, q_f, v')$

Donde v y v' son las cadenas de entrada y de salida respectivamente.

Definición 1.2.6. *Sea M una MT y $(v_1, ..., v_n) \in (X^*)^n$ con $n \in \mathbb{N}$, decimos que M se detiene frente a v si existe una computación de M tal que $c_1 = (\epsilon, q_i, \#v)$, separando cada v_i con un $\#$.*

Con esta definición de computación (de una MT) se tiene cubierto todo lo necesario para describir que funciones pueden ser calculadas usando las MT.

Definición 1.2.7. *Sea $f : (X^*)^n \to X^*$ una función, se dice que f es Turing-Computable si existe una MT M tal que $\forall v = (v_1, v_2, ..., v_n) \in Dom(f)$ se tiene que:*

$$(\epsilon, q_i, \#v_1\#v_2\#...\#v_n) \vdash_M^* (\epsilon, q_f, \#f(v_1, v_2, ..., v_n))$$

Es una computación (siempre que $f(v)$ esta definido).

Regresando al ejemplo 1.2.1 de la función sucesor, ésta técnicamente no es una computación en el sentido de la definición anterior, para que lo sea faltaría poner el cabezal en la primera casilla de la cinta, lo cual de hecho es fácil de realizar (moviendo el cabezal hacia la izquierda hasta encontrar un $\#$).

Consideremos ahora la concatenación de dos máquinas M y M', es decir, el ejecutar la máquina M y con la cadena de salida de M (una vez que la máquina M se detuvo con la entrada dada) ejecutar M'. Esto se puede considerar como solamente ejecutar una MT M'' con alfabeto de cinta $X \cup X'$ y:

$$M'' = \{Y \cup Y', Q \cup Q', \delta \cup \delta', q_i, q'_f\}$$

Ademas M'' debe cumplir que $q'_i = q_f$. Un cosquilleo que puede saltar al instante, puede ser causado por la duda de que pasaría si en M existe un estado q_2 y en M$'$ existe otro estado llamado igual, pero se debe notar que el subíndice 2 (o n $\forall n \in \mathbb{N}$) solo es otra forma de decir que bajo una enumeración le asignamos el 2 como nombre, pero los elementos se pueden considerar diferentes y dar otra enumeración a los estados en M$''$.

Observación. La concatenación de 2 Máquinas de Turing es una MT y por tanto la concatenación finita de máquinas de Turing es una máquina de Turing.

Observación. Composición de funciones Turing-computables es Turing-computable.

1.3. Tesis de Church-Turing

Dado que en los siguientes capítulos veremos algunos modelos de cómputo es necesario introducir un postulado que asegura que todos los modelos construibles son equivalentes (hasta los que no han sido inventados), hasta la fecha

tal enunciado no ha podido ser probado ni refutado.

Tesis de Church-Turing. *Todo procedimiento efectivo (algoritmo) puede ser llevado a cabo por una Máquina de Turing.*

En el enunciado anterior se introduce el concepto de procedimiento efectivo o algoritmo, el cual se define como cualquier procedimiento que pueda ser realizado por algún mecanismo o que existe una sucesión de instrucciones para llevarlo a cabo.

La definición anterior genera algunos problemas, uno de ellos lo vago que puede resultar hablar de mecanismos sin una definición rigurosa de estos (puesto que deben que tener algo de factibles). Otro detalle viene de la noción mas arraigada a la palabra algoritmo ya que una sucesión de instrucciones en realidad no es lo único que permite llevar a cabo la tarea, depende en gran medida quien o que vaya a realizar dicho procedimiento.

Alan Turing ignoró este problema definiendo como algoritmo a las instrucciones que puede llevar a cabo la Máquina de Turing. Todos estos procedimientos se pueden reducir a que funciones se pueden computar con una MT y entonces (considerando que funciones puede computar el mecanismo en cuestión) propuso el siguiente enunciado.

Tesis de Turing. *Si una función f se puede calcular con algún algoritmo entonces f es Turing-computable.*

Alonzo Church propuso un resultado similar pero usando como mecanismo el cálculo Lambda, esto de manera independiente a los resultados de Turing con su MT (por eso que la tesis se llame de Church-Turing) en este caso el enunciado (considerando que el conjunto de las funciones recursivas es la misma que las funciones que Church propuso para dicho cálculo) es el siguiente.

Tesis de Church. *Si una función f se puede calcular con algún algoritmo entonces f es recursiva.*

En el capítulo 4 revisaremos un teorema que muestra (entre otras cosas) que estos enunciados son equivalentes.

La importancia de la tesis cae en el hecho de que no se ha probado de alguna manera su veracidad o falsedad. La veracidad parece no poderse demostrar dado lo vago del concepto de procedimiento efectivo, de hecho se dice que esta hipótesis es de alguna manera un enunciado que sigue en desarrollo, puesto que lo único que se puede hacer al respecto es aportar testigos de ella (modelos computacionales que resultan ser iguales o menos poderosos que la Máquina de Turing). La falsedad del enunciado parece más fácil de comprobar en el papel, solo basta exponer un modelo computacional que pueda computar alguna función que no sea Turing computable, aquí se deja ver la fuerza que ha tomado la Tesis pues tal modelo no se ha propuesto, de hecho todos los modelos presentados hasta la fecha son equivalentes o pueden computar solo un subconjunto de

las funciones Turing-computables.

El objetivo principal de este trabajo es presentar la prueba de que funciones recursivas, Máquina de Turing y Máquina Ábaco (el utilizado en la actualidad en las computadoras personales) son equivalentes en las funciones que pueden computar.

Con el propósito de motivar la lectura de este documento en el camino no solo presentaré dichas pruebas sino que también aparecen varias técnicas y resultados que se utilizan y dan paso en la investigación en el área de la computabilidad actual.

Capítulo 2

Modelos computacionales

Como se mencionó en el capítulo anterior existen varios modelos computacionales y algunos muy diferentes en su funcionamiento, en este capítulo definiré los mas importantes junto con varios ejemplo de funciones que son capaces de computar.

2.1. Notación Modular

Primero es conveniente revisar la notación modular, esta no es mas que una manera de escribir el código de una MT de manera más sencilla, (se puede comparar como el uso de un lenguaje de programación amigable en vez del lenguaje ensamblador para hacer programas en la vida cotidiana) pero por sí misma se puede considerar un modelo computacional, es decir una variante de la MT.

En la siguiente lista se define el significado de cada símbolo (acción) y después mostraremos un ejemplo de alguna máquina escrita en este lenguaje. Cada símbolo representa una MT (de una sola acción), y la concatenación de estos símbolos la concatenación de las MT (excepto las flechas de decisión).

- La flecha izquierda $\triangleleft$ simboliza que el cabezal se mueve a la izquierda.
- La flecha derecha $\triangleright$ simboliza que el cabezal se mueve a la derecha.
- Un caracter del alfabeto de la máquina, simbolizara que la máquina escribe tal símbolo en la casilla actual.
- Las flechas con un caracter del alfabeto indican el flujo del programa, siempre que el caracter sobre la flecha sea el mismo bajo el cabezal.

Por otro lado las flechas no acompañadas por algún carácter son el único camino a seguir (estas se usan solo para la estética del diagrama), además se pueden definir subrutinas y nombrarlas.

Un ejemplo de declaración de subrutinas en este modelo es definir el símbolo $\triangleleft_a$ como la máquina que se mueve a la izquierda hasta encontrar el carácter a y análogamente $\triangleright_a$ a la derecha, esto se puede apreciar mejor en la Figura 2.1, donde hay que notar que el caracter b se usa para abreviar la aparición de una flecha para cada caracter distinto de a.

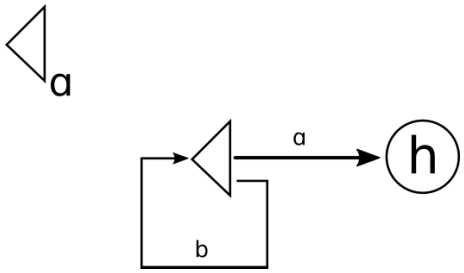

Figura 2.1: Ejemplo de subrutinas.

Las subrutinas anteriores se utilizan frecuentemente en lo que resta de este trabajo. Otra notación extra es que para repetir algún procedimiento n veces se encierra entre paréntesis y eleva a la n (en la notación de la exponencial), un ejemplo de esto se puede observar en la Figura 2.2.

Figura 2.2: Ejemplo de agrupamiento.

Ejemplo 2.1.1. *Máquina que suma (2 elementos).*

Entrada. *La máquina recibe 2 números naturales en la codificación con símbolos $|, \#$ como ya se explicó en el capítulo anterior (Figura 2.3).*

Figura 2.3: Ejemplo de entrada.

Salida. *La suma de dichos números en la misma codificación (Figura 2.4).*

Note que la suma no es la concatenación de símbolos, es decir, la suma de ambos números no es igual a la concatenación de los símbolos $|$ (recuerde que $|$ representa al número 0, por lo que $| + |$ concatenando sería $||$ que representa al 1). Visto esto es fácil convencerse que la máquina en notación modular de la Figura 2.5 realiza la tarea correctamente.

En la notación se agregan 3 detalles extra (que no afectan el funcionamiento), para denotar que una subrutina termina termina se usó la letra r en un circulo, el inicio del programa es el símbolo más arriba a la izquierda (a menos que se

Figura 2.4: Ejemplo de salida.

Figura 2.5: Máquina que suma.

diga cual es explícitamente), y la letra h denota la detención de la máquina. La distinción entre usar h o la r para el final de la ejecución de las máquinas solo se usó para denotar cual es la máquina que termina la tarea completa (la que usa la letra h).

La notación modular no dota de ninguna habilidad extra a las MT (ni le resta ninguna capacidad), para probar esto solo basta ver que cada acción definida en la notación modular tiene su equivalente en las Máquinas de Turing.

Proposición 2.1.1. *Sea M una máquina descrita en notación modular, entonces M es una Máquina de Turing.*

Demostración. Primero hay que fijarse que la letra h encerrada en un circulo describe la MT cuyo único estado es final. De hecho cada símbolo en la lista (como se dijo al definir la notación modular) es una MT en sí, todo esto tiene sentido pues la concatenación de máquinas de Turing es una MT. Con lo anterior dicho lo único que resta es analizar las flechas de decisión que vienen dadas como en la Figura 2.6.

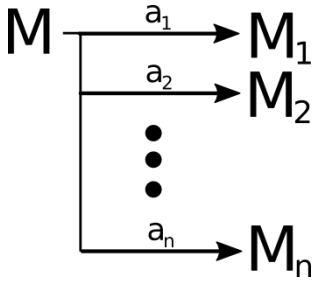

Figura 2.6: Flechas de decisión en notación modular.

En este caso el flujo de la máquina se controla con los estados, es decir, si estás en el estado q de la MT de una acción anterior a la flecha leyendo el caracter a, entonces la función de transición debe estar definida de la siguiente manera $\delta(q, a) = (q', a, \Lambda)$, donde q' es el estado inicial de la MT de una sola acción que sigue a la flecha con una a encima. □

Antes de la siguiente proposición hago la observación que en la notación modular al concatenar 2 símbolos de una sola acción no se toma en cuenta que

caracter esta leyendo el cabezal, esto en la función de transición es equivalente a que $\forall a, b \in Y$ se tenga que $\delta(q, a) = \delta(q, b)$ con q el estado final del primer símbolo concatenado.

Proposición 2.1.2. *Sea M una MT entonces M se puede escribir en notación modular.*

Demostración. Para esta prueba nos fijaremos en δ (la función de transición) y los estados (que controlan el flujo del cómputo). Supongamos que la MT en algún momento de la computación está en el estado q y la función de transición es tal que:

$$\delta(q, a_1) = (q', a_1', A_1)$$

$$\vdots$$

$$\delta(q, a_n) = (q', a_n', A_n)$$

Donde los A_i denotan cualquier movimiento del cabezal (de los 3 posibles) y los a_i junto con a_i' caracteres del alfabeto de Y. La forma de escribir esto se puede hacer como en la Figura 2.6.

Es claro que en esta transformación no se ven las ventajas de la notación modular pues no goza de las abreviaturas, pero hay que recordar que solo es para motivos de la prueba.

Un detalle a tomar en cuenta es que en la notación modular la acción de escribir algún caracter y después mover el cabezal en alguna dirección están separadas siempre, esto no es problema pues se pueden separar las transiciones de este estilo en realizar las acciones una a una en ese orden.

□

Ejemplo 2.1.2. *En este ejemplo se puede ver mejor el proceso de las 2 proposiciones anteriores y además la ventaja de la notación modular. Considerando la máquina que suma de la Figura 2.5, la MT descrita en términos de transiciones queda descrita en la Tabla 2.1.*

q	a	$\delta(q,a)$
q_0	a	$(q_1, a, \triangleright)$
q_1	$\mid$	$(q_1, \mid, \triangleright)$
q_1	$\#$	$(q_2, \#, \Lambda)$
q_2	$\#$	$(q_3, \mid, \Lambda)$
q_3	a	$(q_4, a, \triangleright)$
q_4	$\mid$	$(q_4, \mid, \triangleright)$
q_4	$\#$	$(q_5, \#, \Lambda)$
q_5	a	$(q_6, a, \triangleleft)$
q_6	a	$(q_7, \#, \Lambda)$
q_7	a	$(q_8, a, \triangleleft)$
q_8	a	$(q_9, \#, \Lambda)$
q_9	a	$(q_{10}, a, \triangleleft)$
q_{10}	$\mid$	$(q_{10}, \mid, \triangleleft)$
q_{10}	$\#$	$(q_f, \#, \Lambda)$

Cuadro 2.1: MT que suma, descrita en términos de transiciones

Donde cada linea horizontal, separa cada símbolo de la notación modular.

La discusión anterior solo tiene el propósito de mostrar una manera de describir máquinas de Turing más sencilla, además que la convención usada en la prueba de la proposición anterior (la de separar la acción de escribir algún caracter especifico y mover el cabezal) se utilizará en secciones posteriores.

2.2. Máquina Ábaco

La máquina Ábaco [2] es un modelo computacional importante por su parecido al modelo RAM, además se incluyó por su simpleza aparente y que provee una facilidad para la prueba de que toda función recursiva es Turing-computable. Este modelo está basado en un ábaco real, de hecho es conveniente imaginarlo como tal, con la única diferencia que cuenta con entradas (o líneas) infinitas numerables.

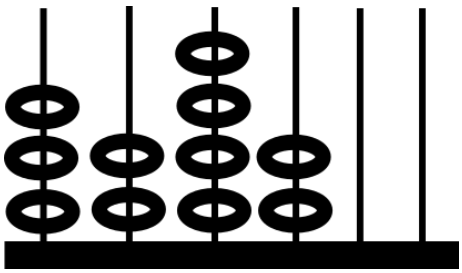

Figura 2.7: Ejemplo de un Ábaco.

La forma en que opera este modelo computacional, es muy simple y a la vez muy poderoso. En cada entrada (llamaremos entrada o registro a una línea con cuentas) se tiene algún número natural y solo permite las operaciones de sumar uno al número en algún registro o restar uno, esto último con la posibilidad de que en caso de restar uno a una entrada que contiene cero esto no se realiza (es decir, no queda -1 en la entrada) y se continua el flujo de la ejecución en otra dirección que en el caso de que tal entrada contara con un natural diferente de cero.

Definición 2.2.1. *Una máquina Ábaco A es una tupla*

$$A = (S, I, \delta, q_i, q_f)$$

tal que:

- $S = \{s : \mathbb{N} \setminus \{0\} \to \mathbb{N} \mid$ *s es una sucesión casi finita* $\}$
- *I es un conjunto finito cuyos elementos los llamaremos instrucciones.*
- $\delta : S \times I \times \mathbb{N} \setminus \{0\} \times \{+, -\} \to S \times I$ *es una función parcial válida para ábaco (concepto que definiré a continuación).*
- q_i *es la instrucción inicial.*
- q_f *es la instrucción final.*

La definición recuerda un poco a la de una máquina de Turing, solo que ahora no hay un cabezal que esté sobre alguna casilla (ni puede operar caracteres directamente), en su lugar δ recibe un natural que denota en que casilla realizará la operación de sumar o restar. Solo queda definir que se entiende por función válida para ábaco.

Definición 2.2.2. *Sea $\delta : S \times I \times \mathbb{N} \setminus \{0\} \times \{+, -\} \to S \times I$ una función parcial con S e I como en la definición de la máquina Ábaco. Se dice que δ es válida para ábaco si:*

- $\delta(s, q_f, n, a)$ *no esta definido para ninguna s,n y a.*
- $\delta(s, q, n, +) = (s', q')$ *con* $s(i) = s'(i)\ \forall i \neq n$ *y* $s(n) = s'(n) - 1$.
- $\delta(s, q, n, -) = (s', q')$ *con* $s(i) = s'(i)\ \forall i \neq n$ *y* $s(n) = s'(n) + 1$, *si* $s(n) > 0$.
- $\delta(s, q, n, -) = (s, q'')$ *con* $q' \neq q''$, *si* $s(n) = 0$.

Describir una máquina Ábaco es sencillo con la notación que se presentará a continuación. Se usa una especie de grafo dirigido donde en cada nodo se denota la posición donde se lleva a cabo la operación seguido del símbolo $+$ ó $-$ denotando cual se realizará. En el caso de que la operación sea exitosa se sigue por la flecha sin símbolo, en otro caso se sigue por una flecha con un punto (veáse la Figura 2.8). La computación comienza con la flecha sin origen y decimos que se detiene si llega a una flecha sin destino.

A continuación se mostrará un ejemplo de una máquina Ábaco.

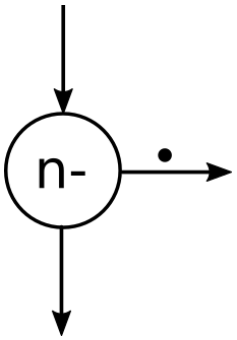

Figura 2.8: Ejemplo de Máquina Ábaco.

Ejemplo 2.2.1. *Máquina copiadora. La máquina descrita en la Figura 2.9 diagrama computa la función* $f : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ *tal que f(a)=(a,a), es decir copia el contenido de la primera casilla en la segunda casilla.*

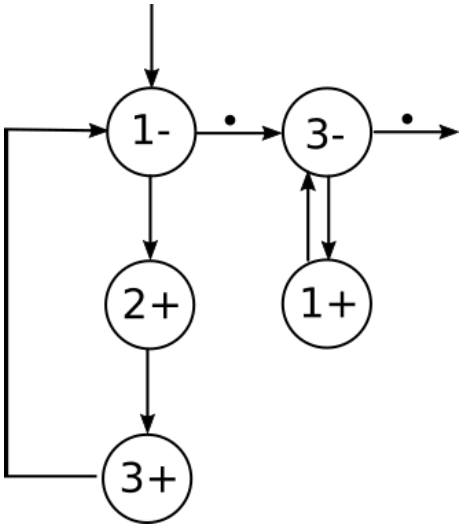

Figura 2.9: Máquina copiadora.

Por convención para considerar que un Ábaco realizó una computación de manera exitosa, los registros extra (aquellos que no se usan para recibir la entrada) se utilizan en orden, es decir, si el Ábaco tiene en uso las primeras 4 entradas de la sucesión y se necesita alguna más, la entrada a usar debe ser la quinta. Esto no resta poder a la máquina pues en realidad a la hora de describir un Ábaco no importa el orden de los registros (solo para escribir el resultado final como se aprecia en la siguiente definición), pero igual la convención es necesaria para facilitar una prueba posterior.

Definición 2.2.3. *Sea $f : \mathbb{N}^n \to \mathbb{N}$ una función, decimos que f es Ábaco-computable si existe un Ábaco A que al recibir como sucesión de entrada a $(a_1, ..., a_n, 0, 0, ..)$ llega al estado final con la sucesión siguiente:*

$$s_f = (f(a_1, ..., a_n), 0, 0, ...)$$

.

Proposición 2.2.4. *Las siguientes funciones son Ábaco-computables.*

- *(La función cero)* $0 : \mathbb{N} \to \mathbb{N}$ *tal que* $\forall n \in \mathbb{N}$ *0(n)=0.*
- *(La función identidad)* $id : \mathbb{N} \to \mathbb{N}$ *tal que* $\forall n \in \mathbb{N}$ *id(n)=n.*
- *(La función sucesor)* $s : \mathbb{N} \to \mathbb{N}$ *tal que* $\forall n \in \mathbb{N}$ *s(n)=n+1.*
- *(Las funciones proyección)* $p_i : \mathbb{N}^m \to \mathbb{N}$ *tal que* $1 \leq i \leq m$ *y* $\forall (a_1, ..., a_m) \in \mathbb{N}^m$ *tenemos que* $p_i(a_1, ..., a_m) = a_i$.

Demostración. La función 0 es computada por la máquina de la Figura 2.8 cambiando la n por un 1.

Para la función identidad es claro que se puede construir un Ábaco cuyo único estado es el estado final.

La función sucesor es computada por un Ábaco con un solo nodo con un 1+.

Para las proyecciones basta ir limpiando las demás casillas (las otras $m-1$) y al final mover el contenido de la $i-ésima$ casilla en la primera casilla. □

Proposición 2.2.5. *Si f y g son funciones Ábaco computables tal que se pueden componer, entonces $g \circ f$ es Ábaco-computable.*

Demostración. Con la convención de que las computaciones hechas por un Ábaco dejan todas las casillas de la sucesión que no forman parte del resultado en 0 solo basta correr un Ábaco (el que computa f) y después correr el Ábaco que computa g con la sucesión de salida del primero como sucesión de entrada del segundo.

Ambas maquinas pueden considerarse un solo Ábaco enumerando las instrucciones nuevamente y uniendo ambas funciones de transición. Esto se realiza análogamente a la concatenación de 2 Máquinas de Turing descrita en la sección anterior. □

En este modelo computacional parece bastante fácil el trabajar con números naturales y parece tener una desventaja clara ante la MT, el hecho de solo poder trabajar con sucesiones de números naturales. Esta idea se perderá en el capítulo 3 y quedará definitivamente olvidada al probar la equivalencia de ambos modelos.

2.3. Funciones recursivas

Las funciones recursivas aparecen frecuentemente en matemáticas, un ejemplo muy común de estas es la serie de Fibonacci que se define como:

$$f(0) = 1$$

$$f(1) = 1$$

$$\vdots$$

$$f(n) = f(n-1) + f(n-2)$$

A grandes rasgos una función en los naturales f es recursiva si su valor en $f(n)$, depende de alguno o varios entre los valores de $f(0), f(1), ..., f(n-1)$. Esta definición aunque es simple no es precisa y necesitaremos hacer una construcción formal de este concepto, esto para poder escribir la prueba de que las funciones recursivas son exactamente las que pueden computar Ábacos y Máquinas de Turing.

Definición 2.3.1. *Las siguientes funciones son llamadas básicas.*

- $0 : \mathbb{N} \to \mathbb{N}$, *la función cero que* $\forall n \in \mathbb{N}$ *0(n)=0.*
- $id : \mathbb{N} \to \mathbb{N}$, *la función identidad.*
- $p_i : \mathbb{N}^n \to \mathbb{N}$, *la proyección en la* $i-$*ésima coordenada que* $\forall (a_1, ..., a_n) \in \mathbb{N}^n$ *tenemos que* $f(a_1, ..., a_n) = a_i$.
- $s : \mathbb{N} \to \mathbb{N}$, *la función sucesor tal que* $\forall n \in \mathbb{N}$ *s(n)=n+1.*

Observación. Las funciones básicas son Ábaco-computables.

La observación anterior como pudo notar es resultado de la proposición 2.2.4.

Es claro (de manera axiomática) que para estas funciones existe un "algoritmo"para calcularlas, es por eso que son consideradas como las funciones base para la construcción de las funciones recursivas.

Ahora se muestran 2 métodos para construir funciones a partir de las funciones básicas, al conjunto de estas funciones (básicas y las construidas con estos métodos) se les denomina funciones recursivas primitivas.

Definición 2.3.2. *Se dice que una función es recursiva primitiva (abreviamos f.r.p.) si se obtiene de alguna de las siguientes maneras.*

- *Una función básica, es f.r.p.*
- *Si g,h son f.r.p. tal que* $g \circ h$ *es función (que la composición tiene sentido), entonces* $g \circ h$ *es f.r.p.*

- *Si* $g : \mathbb{N}^{m+2} \to \mathbb{N}$ *y* $f : \mathbb{N}^m \to \mathbb{N}$ *son f.r.p. (con* $m \in \mathbb{N}$*) entonces la función* $h : \mathbb{N}^{m+1} \to \mathbb{N}$ *definida como sigue es f.r.p.:*

$$h(a_1, ..., a_m, 0) = f(a_1, ..., a_m)$$

$$h(a_1, ..., a_m, a_{m+1} + 1) = g(a_1, ..., a_m, a_{m+1}, h(a_1, ..., a_m, a_{m+1}))$$

Para la última forma de definir funciones recursivas primitivas hay que verificar que en efecto h esta bien definida, pero esto se puede hacer fácilmente por inducción en los números naturales. A las funciones recursivas primitivas definidas por este último método se dice que están definidas por sustitución.

Observación. Todas las f.r.p. son funciones totales.

Proposición 2.3.3. *Las siguientes funciones son f.r.p.*

- $+ : \mathbb{N}^2 \to \mathbb{N}$ *la suma usual en los naturales.*
- $pred : \mathbb{N} \to \mathbb{N}$ *la función predecesor.*
- $- : \mathbb{N}^2 \to \mathbb{N}$ *la resta usual en los naturales, considerando que cuando* $a < b$ *se tiene que* $a - b = 0$*.*
- $* : \mathbb{N}^2 \to \mathbb{N}$ *el producto usual en los naturales.*
- $exp : \mathbb{N}^2 \to \mathbb{N}$ *la exponenciación usual en los naturales.*

Demostración. La suma se puede obtener por sustitución como sigue.

$$+(x, 0) = id(x)$$

$$+(x, s(y)) = s(+(x, y))$$

Cabe resaltar que cada vez que se pruebe la recursividad de alguna operación conocida se utilizará su notación usual por ejemplo, en la suma escribiendo $x+y$ en lugar de $+(x, y)$.

La función predecesor se obtiene por sustitución como sigue.

$$pred(0) = 0$$

$$pred(s(n)) = n$$

La resta queda definida como la suma, pero en términos de la función predecesor.

$$-(x, 0) = id(x)$$

$$-(x, s(y)) = pred(-(x, y))$$

El producto se define en términos de la suma.

$$*(x,0) = 0$$

$$*(x,s(y)) = x + *(x,y)$$

Y la exponenciación en términos del producto.

$$exp(x,0) = 1$$

$$exp(x,s(y)) = x * exp(x,y)$$

□

Existe otra manera de conseguir funciones recursivas que definiré a continuación, con este método las funciones resultantes pueden ser parciales.

Definición 2.3.4. *Sea $f : \mathbb{N}^{n+1} \to \mathbb{N}$ una f.r.p. (con $n \in \mathbb{N}$), se dice que la función $h : \mathbb{N}^n \to \mathbb{N}$ es obtenida por minimización a partir de f si h es tal que (con $\overline{x} \in \mathbb{N}^n$):*

$$h(\overline{x}) = \begin{cases} y & si\ f(\overline{x},y) = 0\ y\ \forall t < y\ tal\ que\ f(\overline{x},t) \neq 0 \\ indefinido & si\ f(\overline{x},t) \neq 0\ \forall t \in \mathbb{N} \end{cases}$$

Cuando defina una función por minimización a partir de una función f, lo denotaremos como min[f].

Definición 2.3.5. *Se dice que una función es recursiva si es f.r.p. o es obtenida por minimización a partir de una función recursiva primitiva.*

Definición 2.3.6. *Sea $f : \mathbb{N}^n \to \mathbb{N}$, f es regular si $\forall(x_1,...,x_{n-1}) \in \mathbb{N}^{n-1} \exists y \in \mathbb{N}$ tal que $f(x_1,...,x_{n-1},y) = 0$.*

Si f es regular entonces la función obtenida por minimización de f es una función total.

Proposición 2.3.7. *Las siguientes funciones son recursivas.*

- $/ : \mathbb{N}^2 \to \mathbb{N}$ *el cociente de la división usual.*
- $res : \mathbb{N}^2 \to \mathbb{N}$ *el residuo de la división usual.*
- $loc : \mathbb{N}^2 \to \mathbb{N}$ *la función que dado el par $a, b \in \mathbb{N}$ dice cuantas veces divide el segundo al primero de manera exacta.*

Demostración. Sea $f : \mathbb{N}^3 \to \mathbb{N}$ definida como sigue:

$$f(a,b,t) = (a+1) - (t+1)b$$

Esta es recursiva pues es composición de f.r.p. Entonces definiendo /=min[f] en efecto esta es la división entera usual, esto se puede comprobar con el algoritmo de la división ya que si $t = /(a,b)$ entonces:

$$(a+1) - (t+1)b \leq 0$$

Lo cual implica que:

$$(a+1) - tb \leq b$$

Y se sigue que:

$$a - tb < b$$

Donde la parte de la izquierda es el residuo de la división el cual debe ser único.

A partir de esto es más fácil definir el residuo como:

$$res(a,b) = a - /(a,b)b$$

Para la función *loc* definiré primero la función $g : \mathbb{N}^3 \to \mathbb{N}$ tal que $g(a,b,t) = 1 - res(a,b^t)$. Se puede ver que g se vuelve 0 si y solo si b^t no divide al número que representa a. Entonces definiendo $loc = min[g] - 1$ se obtiene la función deseada.

□

Al tener definidas que funciones se consideran como recursivas, vale la pena ver la Figura 2.10, en la que quedan descritas las construcciones anteriores.

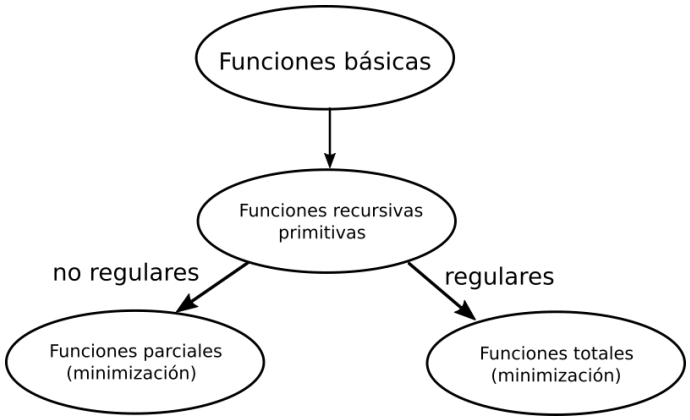

Figura 2.10: Construcción de las funciones recursivas.

Solo falta revisar otra forma de definir funciones recurisvas. A diferencia de la minimización esta no aporta una nueva manera inductiva de hacerlo.

Definición 2.3.8. *Sea $A \subset \mathbb{N}$, se dice que $f : \mathbb{N} \to \{0,1\}$ es su función característica si $f(n) = 1$ si y solo si $n \in A$.*

Definición 2.3.9. *Se dice que un conjunto $A \subset \mathbb{N}$ es recursivo si su función característica es recursiva.*

Nota. Hay que recordar que las relaciones son conjuntos.

En el siguiente teorema se prueba que las funciones definidas por casos son recursivas si las relaciones que definen los casos lo son.

Teorema 2.3.10. *Sean $R_1, ..., R_n$ relaciones recurisvas, de m parametros, disjuntas y $a_1, ..., a_n \in \mathbb{N}$, entonces $f : \mathbb{N}^m \to \mathbb{N}$ definida como sigue es recursiva.*

$$f(x_1, ..., x_m) = \begin{cases} a_1, & si\ R_1(x_1, ..., x_m) \\ \vdots & \vdots \\ a_n, & si\ R_n(x_1, ..., x_m) \end{cases}$$

Demostración. Sean $r_1, ..., r_n$ las funciones características de las relaciones, entonces se puede definir f como sigue:

$$f(x_1, ..., x_m) = \sum_{i=1}^{n} a_i r_i(x_1, ..., x_m)$$

Lo cual es recursivo, pues todas las operaciones involucradas lo son. □

Capítulo 3

Funciones computables y codificación

En este capítulo se trataran dos resultados importantes, el primero que toda función recursiva es Turing-computable pasando por el Ábaco. Este hecho es bastante útil probarlo en este momento, puesto que las funciones que probamos que son recursivas también las podremos considerar (Turing/Ábaco) computables. En segundo lugar se exponen las razones por las cuales las MT no son más poderosas que los Ábacos aun cuando las Máquinas de Turing traten directamente con alfabetos.

Antes de continuar cabe aclarar que solo considero funciones cuyo codominio son los naturales pues las funciones en $\mathbb{N}^m$ con $m \in \mathbb{N}$ se definen coordenada a coordenada.

3.1. Funciones Ábaco-computables

Para probar que toda función recursiva es Ábaco-computable primero se tiene que notar que las funciones básicas son Ábaco-computables, hecho que se probó en la proposición 2.2.4.

Ahora las f.r.p. conseguidas mediante composición son Ábaco-computables, esto se sigue del hecho que la composición de funciones (Ábaco/Turing) computables es computable.

Ahora falta ver que las funciones obtenidas por sustitución y por minimización son Ábaco-computables, hechos que probaré a continuación.

Lema 3.1.1. *Toda f.r.p. es Ábaco-computable.*

Demostración. Sean $f : \mathbb{N}^m \to \mathbb{N}$ y $g : \mathbb{N}^{m+2} \to \mathbb{N}$ funciones recursivas y ábaco-computables con $m \in \mathbb{N}$ y sea $h : \mathbb{N}^{m+1} \to \mathbb{N}$ obtenida por sustitución de f y g. Para computar h hay que considerar el siguiente Ábaco.

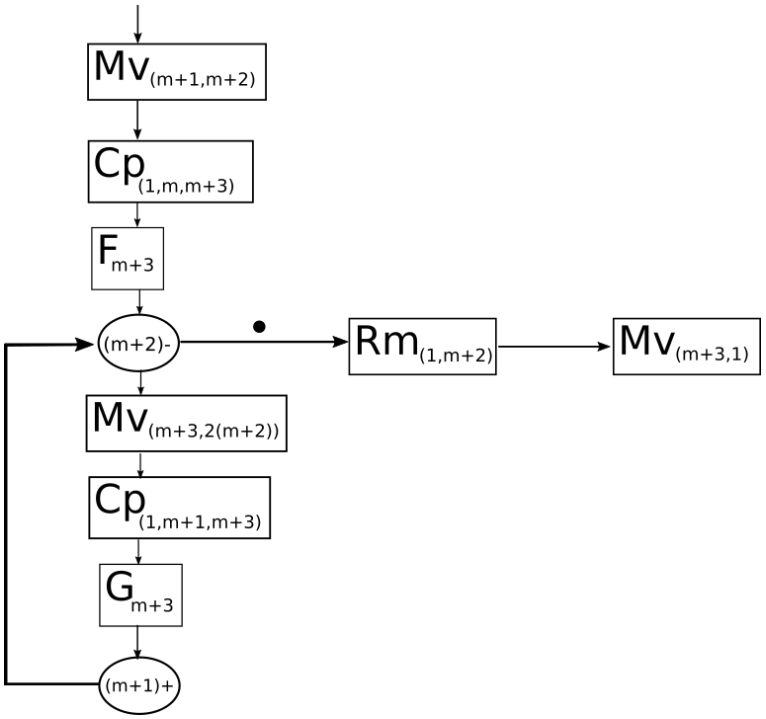

Figura 3.1: Ábaco que computa la sustitución.

Donde F y G son los Ábacos que computan las funciones f y g. El número que tienen de subíndice indica el desfase en los indices de la sucesión, es decir, cada acción de la forma $n+$ y $n-$ se convierte en $(n+m)+$ y $(n+m)-$ (si m es el desfase).

$Cp_{(a,b,c)}$ es el Ábaco que copia el contenido de las entradas entre a y b a las casillas empezando en $s(c)$. $Rm_{(a,b)}$ deja en 0 las entradas entre a y b.Por último $Mv_{(a,b)}$ mueve el contenido de la entrada $s(a)$ a la entrada $s(b)$.

□

Con este resultado probado solo resta la minimización.

Teorema 3.1.2. *Toda función recursiva es Ábaco-computable.*

Demostración. Sea $f : \mathbb{N}^{m+1} \to \mathbb{N}$ recursiva y Ábaco computable, entonces el Ábaco descrito en la Figura 3.2 computa $min[f]$.

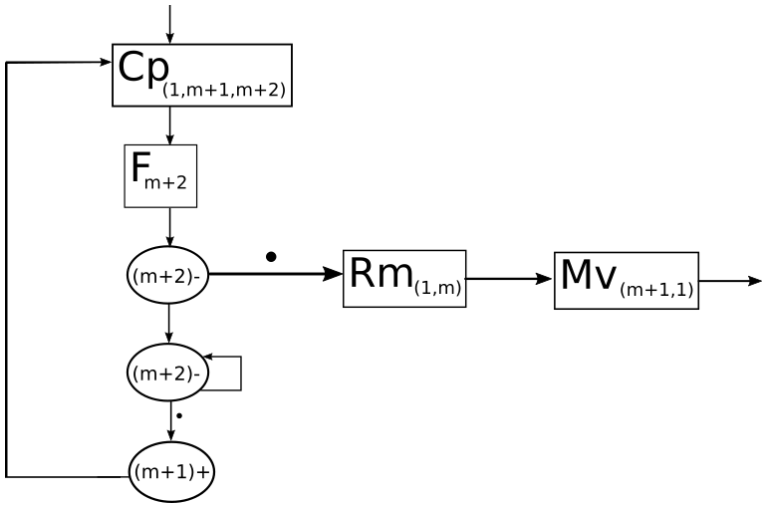

Figura 3.2: Ábaco que computa minimización.

Con esto, el lema anterior, la composición y las funciones básicas cubiertas se tiene el resultado probado. □

Con esto probado todas las operaciones aritméticas (y claro todas las funciones recursivas) que construí de manera recursiva en el capítulo anterior son Ábaco-computables.

3.2. Funciones Turing-computables

En esta sección (análogamente a la sección anterior) probaremos que toda función Ábaco-computable es Turing-computable, esto se logra simulando la ejecución del Ábaco con una MT con el procedimiento que describiré en la prueba.

Teorema 3.2.1. *Sea f una función Ábaco-computable, entonces f es Turing-computable.*

Demostración. Para la representación de los naturales en cada entrada usaremos la ya representación usual de símbolos | pero en las entradas que no son necesarias para la tupla de entrada se rellenan con # (es decir, no se toman en cuenta inicialmente).

Lo que resta es describir como la MT emulara el Ábaco, ésta incluirá como alfabeto de entrada $X = \{|\}$ y $Y = \{|, 0, \#\}$ como alfabeto de la máquina. Comenzando con las acciones de la forma $n+$ como en la Figura 3.4.

Para $n-$ se utiliza la siguiente máquina, donde la flecha con un punto encima solo es para denotar que esas flechas corresponden a sus análogas en el Ábaco (solo para facilidad del lector).

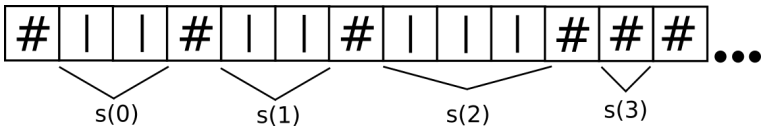

Figura 3.3: Ejemplo entrada.

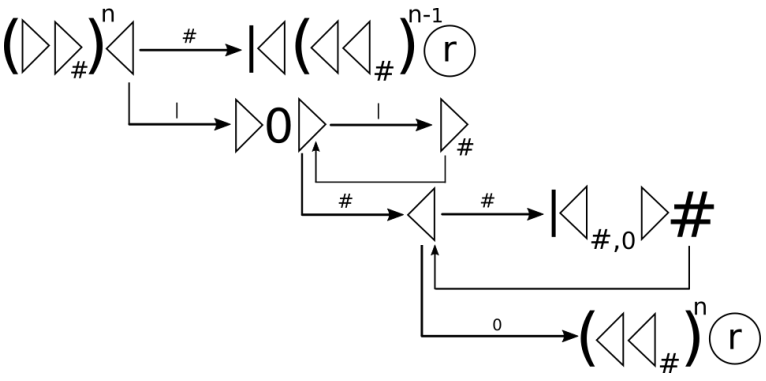

Figura 3.4: Emulando n+.

Aunque en este punto puede parecer que la tarea ha sido completada, falta limpiar las casillas que emulan las entradas que no son $s(1)$. Esto se realiza agregando la máquina de la Figura 3.6 al final.

Ahora en efecto el resultado ha quedado probado.

□

Este resultado junto con el de la sección anterior pudieron probarse más adelante, pero la ventaja (como se dijo al principio del capítulo) de hacerlo en este punto es que las funciones aritméticas que se probaron recursivas en el capítulo anterior ahora se pueden considerar Turing-computables, funciones que se utilizaran en la sección siguiente.

3.3. Codificación de cadenas en los naturales

Comenzaré con la siguiente observación (esto se sigue de que los alfabetos son finitos).

Observación. Cualquier alfabeto es enumerable.

En esta sección se muestran varios resultados que permiten restringirse a solo el estudio de funciones de números naturales entre las funciones Turing-computables lo cual además de acercar el hecho de que los modelos expuestos son equivalentes, también deja ver un proceso de codificación con el cual se

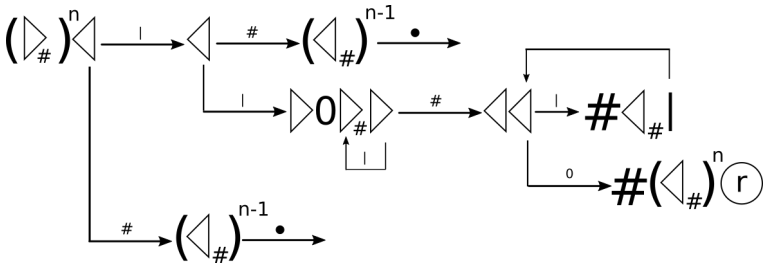

Figura 3.5: Emulando n-.

Figura 3.6: Limpiando la cinta.

pueden manejar cadenas (sobre cualquier alfabeto) indirectamente mediante operaciones aritméticas y enumeraciones computables.

Lema 3.3.1. *Sea Y un alfabeto y $e : \mathbb{N} \to Y$ una enumeración de Y, entonces e es Turing-computable.*

Demostración. Como los alfabetos son finitos se puede literalmente leer el número de entrada n (en la representación ya usual) y escribir el carácter $e(n)$,. □

De la misma manera que el lema anterior se puede probar el siguiente.

Lema 3.3.2. *Sea Y un alfabeto y $e : \mathbb{N} \to Y$ una enumeración de Y, entonces e^{-1} es Turing-computable.*

La siguiente proposición expone en su prueba una codificación que usaremos para las cadenas de un alfabeto.

Proposición 3.3.3. *Sea X un alfabeto, entonces X^* es enumerable.*

Demostración. Sea $e : \mathbb{N} \to X$ una enumeración inyectiva de X tal que $e^{-1}(0)$ es el caracter vacío. Sea $h : X^* \to \mathbb{N}$ tal que:

$$h(a_1...a_n) = \sum_{i=1}^{n} e^{-1}(a_i)|X|^{i-1}$$

Ahora, sean $w_1, w_2 \in X^* \setminus \{\#\}$ cadenas diferentes, por tanto difieren en algún caracter. Hay que considerar la representación de estas cadenas como concatenación de caracteres.

$$w_1 = a_1...a_n$$

$$w_2 = b_1...b_n$$

Si las cadenas no son del mismo tamaño los caracteres sobrantes se pueden considerar vacíos. Sea a_i el primer caracter donde difieren tal que $a_i \neq b_i$. Supongamos h(w_1)=h(w_2), entonces:

$$\sum_{t=i}^{n} e^{-1}(a_t)|X|^{t-1} = \sum_{t=i}^{m} e^{-1}(b_t)|X|^{t-1}$$

De lo cual se deduce que:

$$0 \neq (e^{-1}(a_i) - e^{-1}(b_i))|X|^{i-1} = \sum_{t=i+1}^{m} e^{-1}(b_t)|X|^{t-1} - \sum_{t=i+1}^{n} e^{-1}(a_t)|X|^{t-1}$$

Como los términos de en medio y la derecha son diferentes de 0 se debe cumplir que:

$$(e^{-1}(a_i) - e^{-1}(b_i))|X|^{i-1} < |X|^{i} \leq \sum_{t=i+1}^{m} e^{-1}(b_t)|X|^{t-1} - \sum_{t=i+1}^{n} e^{-1}(a_t)|X|^{t-1}$$

Lo cual es una contradicción. Por tanto h es inyectiva y h^{-1} es la enumeración deseada.

□

Ahora que ya contamos con una enumeración para el conjunto de cadenas que se pueden formar con un alfabeto dado, lo que sigue es ver que este proceso de transformación lo puede llevar acabo una MT.

Proposición 3.3.4. *h^{-1} es Turing-computable.*

Demostración. Primero hay que notar que h^{-1} es una función parcial que devuelve una sucesión casi finita de caracteres (o naturales usando e). El procedimiento para calcular h^{-1} lo describiré a continuación:

1. La MT recibe n, el número de la cadena.
2. Se calcula el residuo de dividir n entre el tamaño del alfabeto (incluyendo el caracter #).
3. A este número se le calcula e y se anota este caracter.
4. Se divide n entre el tamaño del alfabeto (cociente de la división) y se toma el resultado como el nuevo n.
5. El proceso vuelve a empezar desde el paso 2 a menos que n sea 0.

Simplemente hay que notar que todas las operaciones que se utilizan son Turing-computables además que e y su función inversa lo son, pues e es una enumeración finita.

Al final del proceso descrito, en la cinta (después del espacio asignado para n) queda la cadena escrita, se mueve la cadena al inicio y termina la computación (limpiando la cinta). □

Proposición 3.3.5. *h es Turing-computable.*

Demostración. Para esto solo hay que notar que la definición de h hace uso de la suma, la multiplicación y e^{-1} las cuales son todas Turing-computables basta llevarlas a cabo para obtener la imagen de una cadena bajo h. □

Con el hecho de que h y su inversa son Turing-computables entonces solo es necesario considerar funciones que tengan por dominio a $\mathbb{N}^m$ y codominio a $\mathbb{N}^k$ con $m, k \in \mathbb{N}$. Recordemos que no se consideran explícitamente codominios que sean el producto de conjuntos, pues las funciones en estos se definen por coordenadas.

Corolario 3.3.6. *Sea $f : (X^*)^m \to X^*$ una función Turing-computable (con X un alfabeto), entonces existe una función $g : \mathbb{N}^m \to \mathbb{N}$ tal que:*

$$f(w_1, ..., w_m) = h^{-1} \circ g(h(w_1), ..., h(w_m))$$

Con g Turing computable.

Con este hecho probado el estudio de las funciones Turing-computables se reduce al estudio de las funciones de números naturales. Para cerrar esta sección exponemos un lema que permite el uso de solo 3 caracteres, lo cual es de utilidad para probar un resultado del capítulo siguiente.

Lema 3.3.7. *Si M es una Máquina de Turing que computa una función $f : \mathbb{N}^k \to \mathbb{N}$, con alfabeto de cinta X={\|}, entonces existe una MT M' que computa f con alfabeto de cinta X y alfabeto $Y' = X \cup \{\#, 0\}$.*

Demostración. Lo primero que hay que hacer es codificar el alfabeto Y de M, esto con una enumeración e de Y tal que $e(0) = \#$ y $e(1) = |$.

La codificación en este caso será con el número que le toca a cada caracter de Y bajo e pero en representación binaria donde en vez del caracter 0 usamos el caracter #. Esto con el detalle de que todas las representaciones binarias tengan el mismo largo, sea $n \in \mathbb{N}$ este número (por ejemplo si $n = 3$ entonces la codificación de \| es 001).

Antes de que M' empiece a imitar a M necesita codificar su entrada, esto lo realiza con las instrucciones que se muestran en la Figura 3.7.

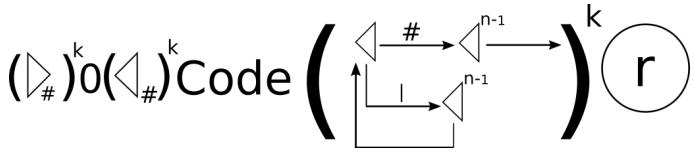

Figura 3.7: Codificación de la entrada por M'.

Donde code es como en la Figura 3.8.

Antes de continuar hay que aclarar que las acciones denotadas como (|/#) se refieren a que la MT escribe el caracter de la izquierda si leyó en la última

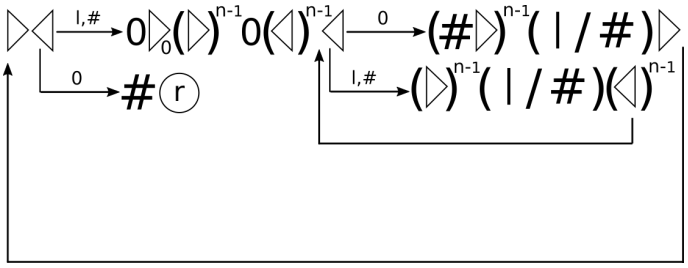

Figura 3.8: Subrutina Code.

flecha de desición el mismo caracter e igualmente escribe el de la derecha si leyó el mismo.

Ahora hay que decir como M' emulara las acciones de M, el acto de mover el cabezal a la izquierda ó derecha cambia a moverse n casillas.

Escribir un caracter pasa a escribir el código de este bajo la codificación, así como leer un caracter pasa a leer el código. Esto último es posible pues el alfabeto es finito.

Para culminar, al terminar la emulación el resultado queda codificado, solo hay que agregar al final de M' las instrucciones de la Figura 3.9.

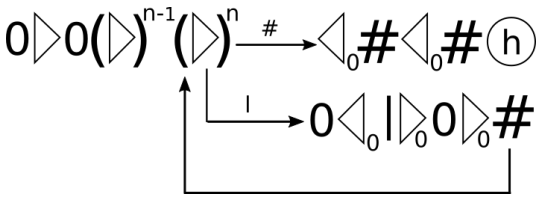

Figura 3.9: Decodificando la cinta de M'.

Con la cinta decodificada M' termina su ejecución.

□

Capítulo 4

Equivalencia de Modelos

Este es el último capítulo, en el mismo terminaremos de exponer la prueba de que los modelos mostrados anteriormente son equivalentes, además de añadir información sobre el problema de la detención.

4.1. Enumeración de las MT

En esta sección (como el título lo indica) se muestra una manera de enumerar las Máquinas de Turing. Este hecho, además de ser una manera un poco rebuscada de decir que solo existen numerables máquinas diferentes (y por tanto funciones Turing-computables), será de utilidad en la siguiente sección.

Lo primero que hay que hacer es fijarse en que a consecuencia del último lema del capítulo anterior el alfabeto de las MT que consideraremos queda reducido a $\{\#, 0, |\}$, símbolos con los que se representan los espacios (#) y los naturales (|).

Con esto se tiene que las acciones posibles de una MT M se pueden enumerar como sigue:

- $0 \rightarrow$ escribir un #.
- $1 \rightarrow$ escribir un 0.
- $2 \rightarrow$ escribir un |.
- $3 \rightarrow$ mover el cabezal a la derecha.
- $4 \rightarrow$ mover el cabezal a la izquierda.

Hay que notar que desde que se definió la notación modular las acciones de escribir y mover se toman en transiciones separadas.

También hay que tomar la convención de que q_0 es el estado final y los estados restantes quedan enumerados desde el 1, junto con la que ninguna transición (omitiendo q_f) $\delta(q, a)$ queda indefinida. Esto es factible pues si se cuenta con

una MT M tal que $\delta(q, a)$ está indefinido se puede rellenar definiendo $\delta(q, a) = (q, a, \Lambda)$ (ciclando la máquina).

Con ambas convenciones tomadas si se tiene una MT M con k estados (sin contar q_f) es posible representar M como una $6k - tupla$ de naturales donde en las casillas pares se encuentre la representación de las acciones (naturales de 0 al 4) y en las impares a que estado pasar (naturales de 0 a k) después de la acción.

Esta representación se encuentra ordenada, es decir, al estado q leyendo un # le corresponde la acción en la casilla 0 (a_1) y pasar al estado descrito en la casilla 1 (b_2); al estado 1 leyendo un 0 la acción en la casilla 2 (a_3) y el estado en la tercera casilla (b_4), y así sucesivamente. Esto se ilustra de mejor forma en la figura 4.1.

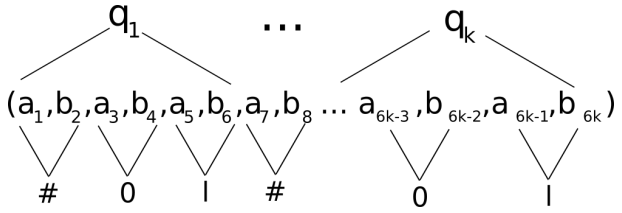

Figura 4.1: Diagrama de la 6k-tupla.

Cabe notar que con este orden si se está en el estado q y leyendo el número i (usando 0 para #, 1 para 0 y 2 para $'|'$) se puede saber en que casilla están las indicaciones que siguen, con las siguientes fórmulas (para la acción a realizar y el estado a pasar respectivamente):

$$6(q-1) + 2i$$

$$6(q-1) + 2i + 1$$

Como extra (al menos para el propósito principal de este texto) cabe mencionar que con esta representación de una MT M como $s \in \mathbb{N}^{6k}$ es posible hacer otra MT M', que dados s y x (una posible entrada para M) simule la ejecución de M, ejecutando las acciones que indique s, anotando el estado en el cual se encuentra M en la cinta y volviendo a la parte de la cinta donde se simula la cinta de M (donde está x inicialmente) para leer el caracter bajo el cabezal simulado y continuar la ejecución.

A una MT como M' se le llama Máquina Universal de Turing, aunque no describimos con detalle el funcionamiento explícito de esta, las ideas detrás de esta máquina que ejecuta máquinas del mismo tipo (la máquina universal también es una MT) son importantes en la teoría de la computabilidad y en la computación aplicada.

Un ejemplo de lo primero (computación aplicada) puede encontrarse en la sección 4.3 donde se expone un poco más sobre el tema y un ejemplo de lo segundo es más fácil de ver, solo basta hacerse la siguiente pregunta ¿usted compra una computadora para cada tarea?.

Volviendo al tema de la enumeración de las MT solo falta considerar una función que nos permita enumerar estas sucesiones finitas, para esto consideremos la función *code* definida en las sucesiones casi finitas de $\mathbb{N}$ como sigue:

$$code(a_1, a_2, ...) = \prod_{i=1}^{\infty} p_i^{a_i}$$

Donde los p_i son los números primos enumerados en orden ascendente. La función está bien definida pues code solo admite sucesiones casi finitas por lo que siempre se cumple que $code(s) < \infty$. Además como *code* es biyectiva en $\mathbb{N} \setminus \{0, 1\}$ es posible recuperar cada valor de la sucesión usando la función *loc* y dividiendo $code(s)$ hasta llegar al 1. Este procedimiento inverso de *code* (función inversa) de hecho es Turing-computable, como se probó en el capítulo 3 (gracias a esto la Máquina Universal podría solo recibir el número de la sucesión bajo *code* y la entrada para ejecutar).

Así a cada máquina en su representación como sucesión le corresponde un natural lo cual otorga una enumeración para el conjunto de las MT.

4.2. Modelos equivalentes

En esta sección probamos el último teorema importante del tema principal de esta tesis (el lector debería estar emocionado). Lo primero que hay que hacer es notar que para una configuración (u, q, v) de alguna MT M, se pueden representar u y v en base 3 (viendo # como 0, | como 1 y $'0'$ como 2) leyendo u hacia la derecha pero leyendo v de izquierda a derecha. Esto se ilustra de mejor forma en la Figura 4.2.

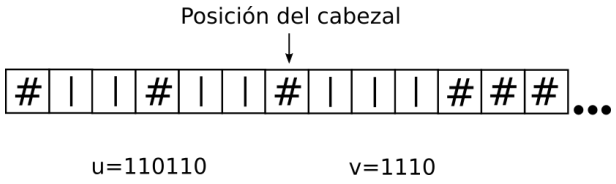

Figura 4.2: Lectura de cinta en base 3.

En el ejemplo (trinToDec convierte de trinario a decimal) $trinToDec(u) = 336$ y $trinToDec(v) = 39$. Con esta forma de representar la cinta se pueden ver las ternas de naturales (l, q, r) como las nuevas configuraciones, donde $l = trinToDec(u)$, q es el número que enumera el estado q y $r = trinToDec(v)$.

Ahora para saber sobre que símbolo se encuentra el cabezal basta calcular.

$$leer(r) = res(r, 3)$$

Para describir las acciones que puede realizar la máquina se usan las siguientes funciones:

- Para escribir en la cinta se usan (Donde a es el número del caracter a escribir).

$$A_0(r,a) = r + a - leer(r)$$

$$A_1(l,a) = l$$

Estas funciones devuelven el nuevo valor a la derecha e izquierda respectivamente.

- Para mover el cabezal a la derecha se usa.

$$A_2(l,r) = 3l + leer(r)$$

$$A_3(l,r) = \lfloor \frac{r}{3} \rfloor$$

Donde A_2 devuelve el nuevo valor de la izquierda y A_3 el de la derecha.

- Para mover el cabezal a la izquierda se usa.

$$A_4(l,r) = \lfloor \frac{l}{3} \rfloor$$

$$A_5(l,r) = 3r + leer(l)$$

Donde A_4 devuelve el nuevo valor de la izquierda y A_5 el de la derecha.

Así en general se pueden considerar las siguientes funciones.

$$nr(l,r,i) = \begin{cases} A_0(r,i) & \text{si } i = 0 \text{ ó } i = 1 \text{ ó } i = 2 \\ A_3(l,r) & \text{si } i = 3 \\ A_5(l,r) & \text{si } i = 4 \end{cases}$$

$$nl(l,r,i) = \begin{cases} A_1(l,i) & \text{si } i = 0 \text{ ó } i = 1 \text{ ó } i = 2 \\ A_2(l,r) & \text{si } i = 3 \\ A_4(l,r) & \text{si } i = 4 \end{cases}$$

Las cuales permiten, dada la configuración actual y la acción a realizar, conocer la nueva configuración de la cinta (sin considerar el estado).

A este proceso evidentemente le falta el uso de los estados para saber que acción realizar y a que estado pasar. Ahora usando la codificación de la sección anterior de una MT en $6k - tupla$ se tiene que dado el estado actual q y el caracter bajo el cabezal $leer(r)$ se obtienen 2 funciones que devuelven la acción a realizar y el nuevo estado.

$$nEstado(m,q,r) = p_m(6(q-1) + 2leer(r))$$

$$sAccion(m,q,r) = p_m(6(q-1) + 2leer(r) + 1)$$

Donde $p_m(i)$ te da la proyección en la $i-ésima$ coordenada de la $6k-tupla$ de la $m-ésima$ MT.

El paso siguiente consiste en describir las configuraciones en el paso t de la ejecución, es decir, cual es la configuración de la cinta en la $t-ésima$ transición. Antes de esto es conveniente codificar las configuraciones en los naturales como sigue:

$$pConf(l, q, r) = 2^l 3^q 5^r$$

Además como $pConf$ es inyectiva es posible recuperar l, q y r con la función loc.

Volviendo a la configuración de la MT dado el paso t, la función deseada puede definirse por sustitución como sigue:

$$conf(m, x, 0) = pConf(0, 1, x)$$

$$conf(m, x, s(t)) = pConf(nl(l, r, i), q', nr(l, r, i)))$$

Donde x es la entrada de la máquina y q', l, r e i se obtienen de la siguiente lista (donde cA es la configuración actual):

- $conf(m, t) = cA$
- $loc(cA, 2) = l$
- $loc(cA, 3) = q$
- $loc(cA, 5) = r$
- $nEstado(m, q, r) = q'$
- $sAccion(m, q, r) = i$

A estas alturas de la codificación ya es pertinente enunciar el siguiente teorema.

Teorema 4.2.1. *Sea $f : \mathbb{N}^k \to \mathbb{N}$ una función Turing computable entonces f es recursiva.*

Demostración. Sea M la MT que computa f y m el número de M (en la enumeración de la sección anterior). Entonces cabe notar que M se detiene en posición estándar cuando es verdadero que:

$$(l = 0) \wedge (q = 0) \wedge (r = trinToDec(\#f(\overline{x})))$$

Con $\overline{x} \in \mathbb{N}^k$ la representación no codificada de la entrada de M.

Sea $nstd(l, q, r, f(\overline{x}))$ la función característica del conjunto que describe la negación de la fórmula lógica anterior.

Entonces M se detiene en posición estándar sí y solo sí:

$$nstd(conf(m, x, t), f(\overline{x})) = 0$$

Aunque usar $conf$ como argumento es un abuso de notación estamos haciendo referencia usar loc para recuperar l, q, y r de $conf$.

Así el valor de $f(\overline{x})$ puede ser obtenido por la función:

$$salida(m, x, t) = cant(loc(conf(m, x, t), 2))$$

Donde $cant(r)$ cuenta la cantidad de símbolos | en la representación trinaria de su argumento definida como sigue:

$$cant(r) = log_2\left(\frac{r}{2} + 1\right)$$

Esta definición de $cant$ tiene sentido pues en el momento que se necesita (si t es tal que la máquina se detuvo en posición estándar) se tiene que v es un # seguido de $f(\overline{x})$ símbolos |.

Ahora para conseguir tal t se define:

$$det(m, x) = min[nstd](m, x, t)$$

Antes de terminar es preciso resaltar que el valor $f(\overline{x})$ se puede obtener corriendo M con entrada x.

Así $F(m, x) = salida(m, x, det(m, x))$ devuelve $f(\overline{x})$ el valor final de la computación y como las MT son enumerables se puede ver a m como el natural de F bajo la enumeración y así se tiene que $f(x)$ es una función recursiva.

Aunque no se mencionó en cada paso, cada función que compone $F(m, x)$ es recursiva por lo que la prueba tiene sentido. □

Gracias al teorema anterior se puede enunciar el siguiente resultado.

Corolario 4.2.2. *Sea $f : \mathbb{N}^k \to \mathbb{N}$ una función, entonces las siguientes afirmaciones son equivalentes:*

- *f es Turing-computable.*
- *f es Ábaco-computable.*
- *f es recursiva.*

Este resultado junta 2 conceptos de algoritmo que en cierto modo son distintos. Por un lado la Máquina de Turing conserva el concepto de programación más común. Por otro las funciones recursivas son constructivas (inductivas) y se debe tener su funcionamiento planeado antes de iniciar su construcción.

4.3. El problema de la detención y la computabilidad

Esta sección en cierto modo puede considerarse extra, pero la incluimos por la importancia que tiene. En la sección 3.3 se mecionó la Máquina Universal de Turing la cual se describe como una MT que dado el número de otra máquina y una entrada para esta es capaz de simular su ejecución con el uso de la $6k-tupla$. Como ya se dijo en esa misma sección no se describirá a detalle el funcionamiento de esta, pero su existencia abre la posibilidad de probar (de una manera) que no todas las funciones son Turing-computables (de números naturales, pues la afirmación en general es cierta dado el hecho que las funciones Turing-computables son númerables), lo cual aceptando la tesis de Church-Turing (presentada en la sección 1.3) implica que hay problemas para los cuales no existe un algoritmo (por más lento que fuera) para resolverlo.

El problema de la detención no expone la única función que no es computable, pero este es el ejemplo más común y de fácil entendimiento, además que abrió la puerta al estudio de la computabilidad con el enfoque que se dedica a clasificar que tan difíciles son los problemas que no son computables.

Básicamente el problema consiste en dada una máquina con número m (en la enumeración ya expuesta) y entrada x ¿la máquina terminará su ejecución?.

Supongamos que existe una MT M tal que recibe un número natural n (correspondiente a una MT en la enumeración) tal que:

$$M(n) = \begin{cases} \text{indefinido} & \text{si } n \downarrow (n) \\ 0 & \text{si } n \not\downarrow (n) \end{cases}$$

Aquí se hace introducción de esta notación, como consideramos máquinas que computan funciones, si M computa f en vez de escribir $f(x)$ se puede cambiar por $M(x)$. También $n \downarrow (x)$ significa que la MT con número n se detiene con entrada x, asi como $n \not\downarrow (x)$ significa que la ejecución no termina.

Volviendo al tema, si existiera una MT capaz de decir si otra MT se detiene con alguna entrada esta se puede modificar para conseguir M, ciclando la ejecución en caso de que responda que la máquina se detuvo e imprimiendo 0 en caso de que la máquina no termine.

Ahora solo hay que fijarse en que ocurre si M recibe su propio número de máquina m.

$M(m) = indefinido$. Esto implica que $m \downarrow (m)$ lo cual es una contradicción.

$M(m) = 0$. Esto implica que $m \not\downarrow (m)$ lo cual es una contradicción.

Por tanto no existe una MT que pueda decidir si otra máquina no se detendrá con cierta entrada. Este es un ejemplo de una función que no es Turing-computable, este hecho (la existencia de funciones que no son computables por alguna MT) abre paso al estudio de la computabilidad basada en grados de

Turing. A grandes rasgos el estudio en esa área se encarga de clasificar los problemas que no son computables por grado de dificultad, dando dificultad cero a los problemas computables.

La base de esta clasificación proviene de la pregunta de que problemas se podrían resolver (que funciones se podrían computar) si existiera en efecto una máquina que compute el problema de la detención, la respuesta a la que se ha llegado es que no todos.

Este tipo de estudios tienen base (al menos en propósito) en la Tesis de Church-Turing pues de existir un modelo computacional construible que computara más funciones que la MT, por ejemplo la detención, esta jerarquización tendría algunos cambios al menos en la enumeracion de la dificultad asignada a los problemas.

Esto no debe ser confundido con la complejidad computacional, es decir, los problemas llamados no polinomiales (y sus variantes) que le competen a otra rama de las ciencias de la computación donde dividiendo a los problemas en polinomiales, no polinomiales, no polinomiales completos, etc. se busca estudiar la rapidez de los algoritmos. Pero todos los problemas estudiados en esta rama son de grado 0 (son computables).

Conclusión

A lo largo de este trabajo además de presentar los elementos necesarios para realizar la prueba de los resultados propuestos hemos visto algunas técnicas muy útiles, empleadas en las ciencias de la computación y la lógica.

Con respecto a la equivalencia de modelos computacionales, hasta la fecha no se conoce algún modelo que pueda realmente computar más funciones que las que conforman el conjunto de las funciones recursivas. Este hecho solo refuerza la tesis de Church-Turing, al menos a modo de creencia (en el sentido de tenerle fe), pues tampoco existe una prueba del enunciado, en parte porque el enunciado no se encuentra dentro de una teoría matemática. Se han hecho inclusiones dentro de teorías, pero al fijar el significado de algoritmo (procedimiento efectivo, proceso realizado por una máquina, etc) se pierde parte del significado de este.

Las pruebas como las de este texto, o las que involucran otros modelos computacionales han funcionado muy bien como testigos de la tesis, ademas de mostrar que la aritmética (estándar) forma parte esencial en la búsqueda de fijar que se puede computar (para algún modelo nuevo por ejemplo).

Desde mi punto de vista los esfuerzos de encontrar un contra ejemplo se han desviado, dada la cada vez más fuerte aceptación que tiene el enunciado. Aún así los estudios de la computabilidad continúan (como se mencionó al final de la sección 4.3).

Bibliografía

[1] Fernando Hernández Hernández, *Teoria de Conjuntos una introducción*, Instituto de Matemáticas, UNAM, Sociedad Matemática Mexicana, Segunda edición, 2003.

[2] George S. Boolos, John P. Burgess, Richard C. Jeffrey, *Computability and Logic*, Cambridge University Press, Fifth edition, 2007.

[3] Gonzalo Navarro, *Teoria de la computación (Lenguajes Formales, Computabilidad y Complejidad)*, Notas, 2014.

[4] Douglas S. Bridges, *Computability A Mathematical Sketchbook*, Springer-Verlag New York, First edition, 1994.

[5] Julio Ernesto Solis Daun, Yolanda Torres Falcón *Lógica Matemática*, Universidad Autónoma Metropolitana Unidad Iztapalapa, Primera edición, 1995.

[6] Antonio Montalbán *Matemática Computable*, Notas, 2009.