# COMPLEJIDAD COMPUTACIONAL Y ÁLGEBRAS DE FUNCIONES

ALBERTO ROMERO GONZÁLEZ


Trabajo Fin de Grado

Dirigido por Francisco Félix Lara Martín

Dpt. de Ciencias de la Computación e Inteligencia Artificial
Facultad de Matemáticas
Universidad de Sevilla

Septiembre 2016

Alberto Romero González: *Complejidad computacional y álgebras de funciones,* Trabajo Fin de Grado, Septiembre 2016

## ABSTRACT

Usually, computational complexity classes are given explicitly using computation models and certain restrictions on available resources (time and/or space). However, in many cases, it is possible to obtain alternative descriptions of these classes as algebras of functions.

In this work, some results of this type are presented. Logarithmic and lineal time hierarchies are characterized by functions algebras. We also present some functions algebras for polynomial time, logarithmic space and linear space complexity classes.



## RESUMEN

Las clases de complejidad computacional suelen darse de manera explícita mediante modelos de computación y ciertas restricciones sobre los recursos disponibles (normalmente tiempo y/o espacio). Sin embargo, en muchos casos, es posible obtener descripciones alternativas de dichas clases como álgebras de funciones.

En este trabajo se presentan algunos resultados de este tipo, caracterizando mediante álgebras de funciones las jerarquías de tiempo logarítmico y tiempo lineal, y las clases de complejidad de tiempo polinomial, espacio logarítmico y espacio lineal.

# ÍNDICE GENERAL

# 1

# INTRODUCCIÓN.

Tradicionalmente, encontrar un procedimiento práctico que dé solución a un problema concreto ha concentrado gran parte del esfuerzo y desarrollo de las matemáticas. Estos procedimientos se basaban en una noción intuitiva de algoritmo, dejando de lado la tarea metamatemática de encontrar una formalización precisa de la noción de algoritmo.

Es a principios del siglo pasado cuando, en el campo de la lógica matemática, surge la preocupación por formalizar conceptos como prueba, algoritmo o función computable. En la década de 1930, trabajos como los de Church, Gödel o Turing que dan una definición robusta y precisa de qué significa que un problema sea computacionalmente resoluble (i.e. decidible), utilizando para ello diversos modelos computacionales. Esto lleva aparejado una primera agrupación de problemas en dos clases: decidibles o no decidibles.

En lógica es muy común definir una clase de manera inductiva, es decir, aportando unos elementos iniciales de la clase y dando después unos operadores que produzcan nuevos elementos de la clase. Bajo esta idea subyace la noción de álgebra de funciones: la menor clase de funciones que contiene ciertas funciones iniciales y es cerrada bajo ciertos operadores. Por ejemplo, esta formalización es utilizada por Gödel en la definición de funciones recursivas primitivas, Church en el $\lambda$-cálculo o Kleene en la funciones $\mu$-recursivas.

A mediados del siglo XX, la comercialización de las primeras computadoras empuja hacia el nacimiento de la Teoría de la Complejidad. Interesa ahora conocer, además de si un problema es decidible o no, si la solución a dicho problema es "practicable" teniendo en cuenta la limitación de ciertos recursos usados en el cálculo de la solución. De esta forma, nace la necesidad de clasificar los de problemas en base al consumo que la computación de la solución de los mismos realiza de ciertos recursos prefijados, normalmente tiempo y espacio.

En 1963, Robert W. Ritchie prueba que la clase $\mathcal{E}^2$ de Grzegorczyk se corresponde con la colección de funciones computables en espacio lineal y, en 1965, Alan Cobham caracteriza las funciones computables en tiempo polinomial como un cierto un álgebra de funciones bajo un esquema de recursión. Estos resultados, además de suponer el inicio de la Teoría de la Complejidad, aportaron técnicas de aritmetización que condujeron a encontrar diversas algebras de funciones que se identifican con otras clases de complejidad.

De lo anterior, se extrae, además, una importante consecuencia que forma parte de los objetivos de este trabajo: las caracterizaciones dadas para las clases complejidad son independientes del modelo de computación en que éstas han sido originalmente definidas.

Típicamente, las clases de complejidad se definen explícitamente a través de un modelo computacional, una colección de recursos y una función conocida

que sirve de cota para estos recursos. Ocurre que, aunque los modelos computacionales a veces estén basados en ideas bastante intuitivas, estudiar dichas clases de complejidad directamente sobre estos modelos puede convertirse en tarea tan abstracta como farragosa. Es por esto, que conseguir descripciones alternativas de las clases de complejidad puede ayudar a describir las mismas y conseguir un mejor conocimiento sobre las jerarquías en que éstas se estructuran.

El objetivo central de este trabajo es presentar algunas clases de complejidad relevantes y mostrar con cierto detalle cómo éstas pueden ser caracterizadas por un álgebra de funciones dado. Concretamente, se siguen los resultados del artículo de Peter Clote [3] para demostrar minuciosamente cómo un álgebra de funciones concreto se corresponde con la jerarquía de complejidad de tiempo logarítmico. Posteriormente, se utilizan y adaptan las técnicas usadas en el caso anterior para probar que ciertas álgebras de funciones son equivalentes a las clases de complejidad de tiempo polinomial, de espacio logarítmico y de espacio lineal, y, finalmente, a la jerarquía de tiempo lineal.

El contenido de este trabajo se estructura en dos partes diferenciadas. En primer lugar, en los capítulos 2 y 3 presentamos las nociones y herramientas necesarias para sustentar el resto del trabajo. Específicamente, en el capítulo 2 se introducen los modelos de computación sobre los que se van a definir las clases y jerarquías de complejidad a estudiar. Todos ellos son diferentes máquinas de Turing básicas (con o sin acceso aleatorio), concretamente: máquinas de Turing Deterministas, No Deterministas y Alternantes. En el capítulo 3, se especifican las nociones de aceptación de un lenguaje, así como diferentes clases y jerarquías de complejidad que se basan en los modelos computacionales dados en el capítulo anterior. Además, se define la noción de álgebra de funciones y cómo el cálculo de las funciones que componen dichas álgebras puede ser simulado por máquinas de Turing, bajo ciertas condiciones.

Una segunda parte estaría compuesta por los capítulos 4 y 5. En el primero de ellos se describen los elementos y propiedades de un álgebra de funciones concreto, $A_0$, para, posteriormente, mostrar que es posible codificar la computación de una máquina de Turing de acceso aleatorio en dicho álgebra. Finaliza probando la equivalencia entre ésta y la jerarquía de funciones de tiempo logarítmico, LH. Las herramientas y técnicas utilizadas se muestran de manera pormenorizada, ya que servirán de apoyo para el capítulo siguiente, en el cual, se prueba que las álgebras de Cobham y Lind (adaptadas para enteros) caracterizan las clases de funciones de tiempo polinomial, PTIME, y espacio logarítmico, LOGSPACE, respectivamente, y que las álgebras $\mathcal{E}^2$ y $\mathcal{M}^2$ presentadas por Grzegorczyk se corresponden con las clases de complejidad de espacio lineal, LINSPACE y la jerarquía de tiempo lineal, LTH, respectivamente.

# 2

# MODELOS DE COMPUTACIÓN.

Al lo largo de este capítulo introducimos algunos modelos de computación clásicos que van a servir para determinar distintas clases de complejidad. Profundizar en su detalle permitirá, en capítulos posteriores, dar sobre estos modelos las funciones iniciales y operadores de cierre de ciertas álgebras de funciones que caracterizarán las clases de funciones computables correspondientes.

## 2.1 MÁQUINAS DE TURING.

En 1936 Alan M. Turing introduce estos modelos como herramienta en su estudio del Problema de decisión para la Lógica de primer orden propuesto por Hilbert. Se trata de una descripción abstracta de un dispositivo mecánico que intenta imitar el comportamiento humano ante un cálculo aritmético. Siguiendo esta idea, una máquina de Turing está compuesta por una cinta, infinita por la derecha, subdividida en celdas que contienen un único símbolo. En un momento dado del cálculo, el humano estará observando el símbolo escrito en una sección concreta de la cinta y se encontrará en una cierta "condición mental" (*estados* de la máquina). En base a estos dos hechos, actuará conforme a una lista finita de opciones que conoce previamente, pudiendo así cambiar la porción de cinta que observa y/o su estado mental.

A lo largo de esta sección se irán añadiendo ciertas modificaciones a estas máquinas de Turing básicas para posibilitar la definición y delimitación de algunas clases de complejidad, aunque, en la mayoría de los casos, estos modelos serán equivalentes.

De la descripción informal anterior se desprende que dichas máquinas procesan cadenas de símbolos de un alfabeto finito y de esto, que el hecho de realizar un cálculo sobre números es en realidad procesar una cadena de símbolos mediante un algoritmo. Será necesario pues, comenzar dando algunos conceptos sobre estas "cadenas de símbolos":

**Definición 2.1.1.** *Un* alfabeto, $\Sigma$ *es un conjunto finito, no vacío, de objetos denominados* símbolos.

**Definición 2.1.2.** *Una* palabra *o* cadena *es una* $n$*–tupla de símbolos de una alfabeto* $\Sigma$*, siendo* $\varepsilon$ *la palabra vacía. Normalmente se denota a estas* $n$*–tuplas de la forma* $a_1a_2\cdots a_n$ *en lugar de* $(a_1,\ldots,a_n)$*. Además, si* $u = a_1a_2\cdots a_n$*, se dice que la* longitud *de la palabra* $u$ *es* $n$ *(se nota* $|u| = n$*).*

**Definición 2.1.3.** $\Sigma^*$ *será el conjunto de todas las palabras sobre el alfabeto* $\Sigma$*.*

Con estas nociones se está en disposición de describir formalmente algunos tipos de máquinas de Turing.

### 2.1.1 *Máquinas de Turing multicintas deterministas.*

Una máquina de Turing multicinta es el resultado de ampliar el concepto de máquina de Turing dado en el inicio de esta sección añadiéndole un número finito de cintas de trabajo sobre las que se pueden leer, escribir y borrar símbolos. Para definir este modelo será necesario dar uno o más alfabetos junto con un símbolo especial (notado por $B$) que no pertenecerá a los mismos y que representará el carácter blanco, es decir, el hecho de que en la cinta aparezca una celda en blanco.

**Definición 2.1.4.** *Una* máquina de Turing multicinta determinista *(en adelante* TM*)* $M$ *queda definida por la 6-tupla* $(Q, \Sigma, \Gamma, \delta, q_0, k)$ *donde* $k \in \mathbb{N}$ *y:*

- $Q$ *es un conjunto finito cuyos elementos llamamos* estados*. Dentro de este conjunto se pueden diferenciar ciertos estados:*
  - *Estado inicial* $q_0$*. Será el estado desde el que se parta cada vez que la máquina se ejecute sobre una palabra.*
  - *Estado de aceptación* $q_A$*.*
  - *Estado de rechazo* $q_R$*.*
- $\Sigma$ *es un alfabeto finito de sólo lectura.* $\Sigma \cup \{B\}$ *contiene los símbolos que podremos leer en la cinta de entrada de la máquina, siendo* $B$ *un carácter auxiliar que llamamos carácter blanco.*
- $\Gamma$ *es un alfabeto finito de lectura/escritura.* $\Gamma \cup \{B\}$ *será el alfabeto usado en las cintas de trabajo.*
- $\delta$ *es una función a la que llamamos* de transición *tal que:*

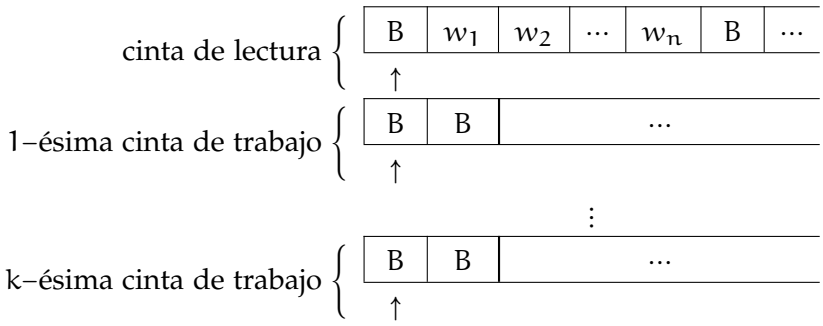
$$\delta : (Q - \{q_A, q_R\}) \times (\Sigma \cup \{B\}) \times (\Gamma \cup \{B\})^k \longrightarrow Q \times (\Gamma \cup \{B\})^k \times \{-1, 0, 1\}^{k+1}.$$

Esta definición se entiende como una abstracción de una máquina de Turing multicinta en la que se precarga una palabra precedida del carácter $B$ en la cinta de lectura y el resto (las $k$ cintas de trabajo) se mantienen en blanco. Cada cinta, que es infinita por su lado derecho, tiene una cabeza de lectura que puede:

1. Leer el símbolo de la celda en la que se encuentra.
2. En el caso de las cintas de trabajo, también puede escribir un símbolo en la celda a la que apunta.
3. Moverse una celda hacia la derecha, una celda hacia la izquierda o permanecer en la misma celda.

**Definición 2.1.5.** *Una* configuración *de la máquina de Turing será un elemento de* $Q \times (\Sigma \cup \{B\})^* \times (\Gamma \cup \{B\})^{*k} \times \mathbb{N}^{k+1}$ *simbolizando:*

$$\underbrace{Q}_{\text{Estado}} \times \underbrace{(\Sigma \cup \{B\})^*}_{\text{Contenido cinta lectura}} \times \underbrace{(\Gamma \cup \{B\})^{*k}}_{\text{Contenido cintas trabajo}} \times \underbrace{\mathbb{N}^{k+1}}_{\text{Posición de las } k+1 \text{ cabezas}}$$




Cuadro 1: Configuración inicial de una máquina de Turing multicinta.

De esta forma, las configuraciones indican el estado actual en el que se encuentra la máquina, el contenido de las cintas y la posición de las cabezas. Para abreviar, se representa la posición de la cabeza de una determinada cinta simplemente subrayando el símbolo que se encuentra leyendo en ese momento concreto. Por ejemplo, una posible configuración podría ser $(q, Bb\underline{a}bbaB, Ba\underline{b}B, B\underline{a}bB)$ donde la máquina se encuentra en un estado $q$, la cabeza de la cinta de lectura se encuentra leyendo la segunda $b$, y las cabezas de la cintas de trabajo se encuentran leyendo los símbolos $b$ y $a$, respectivamente.

**Definición 2.1.6.** *Una configuración $\alpha = (q, \ldots)$ será de* parada *si su estado $q$ pertenece al conjunto $\{q_A, q_R\}$. En otras palabras, si el estado de la configuración $\alpha$ es de aceptación o de rechazo.*

En el proceso de describir el funcionamiento de una máquina de Turing, el siguiente paso será estudiar las condiciones necesarias para que una configuración $\beta$ sea la siguiente de otra $\alpha$. Supongamos entonces que $\alpha$ y $\beta$ son de la forma:

$$\begin{aligned} \alpha &= (q, BxB, \alpha_1, \ldots, \alpha_k, n_0, n_1, \ldots, n_k) \\ \beta &= (r, BxB, \beta_1, \ldots, \beta_k, m_0, m_1, \ldots, m_k) \end{aligned}$$

donde:

- La celda $n_0$ de la cinta de entrada (en que está cargada la palabra $BxB$) contiene el símbolo $a$.
- Para todo $i \in \{1, \ldots, k\}$ se tiene:
  a) $\alpha_i = u_i \sigma_i v_i$ y $\beta_i = u_i \tau_i w_i$ donde $\sigma_i, \tau_i \in \Gamma \cup \{B\}$ y $u_i, v_i, w_i \in (\Gamma \cup \{B\})^*$.
  b) $|u_i| = n_i$. Acordamos que la celda más a la izquierda de una cinta es la $0$–ésima celda y la $n$–ésima celda es la que tiene $n$ celdas a su izquierda. Se entiende entonces que $\sigma_i$ y $\tau_i$ serán el contenido de la $n_i$–ésima celda de la $i$–ésima cinta de las configuraciones $\alpha$ y $\beta$ respectivamente.

**Definición 2.1.7.** *Con las condiciones anteriores, se dice que* $\beta$ *es la* configuración sucesora *de* $\alpha$ *(notado* $\alpha \vdash_M \beta$*) en una máquina de Turing* $M$ *sobre una palabra* $x$ *si:*

$$\delta(q, a, \sigma_1, \ldots, \sigma_k) = (r, \tau_1, \ldots, \tau_k, m_0 - n_0, m_1 - n_1, \ldots, m_k - n_k)$$

*cumpliendo para todo* $i \in \{1, \ldots, k\}$ *lo siguiente:*

*a)* $m_i < |\beta_i|$.

*b) Se da una de estas dos opciones:*

- $v_i = w_i$.
- $v_i = \varepsilon$, $w_i = B$ *y* $m_i = n_i + 1$.

**Definición 2.1.8.** *Una* computación *de una máquina de Turing* $M$ *es una sucesión, no necesariamente finita, de configuraciones* $C_1, \ldots, C_n, \ldots$ *tal que, para cada* $i \in \{1, \ldots, n, \ldots\}$*, se tiene que* $C_i \vdash_M C_{i+1}$.

*Si* $C_1, \ldots, C_n$ *es una computación finita escribiremos* $C_1 \vdash_M^n C_n$.

*Diremos que una computación finita* $C_1, \ldots, C_n$ *es de* aceptación *si el estado de* $C_n$ *es* $q_A$.

**Definición 2.1.9.** *Una máquina de Turing* $M$ *acepta un lenguaje* $L \subseteq \Sigma^*$*, denotado* $L = L(M)$*, si para toda palabra* $w \in L$ *existe una computación de aceptación cuya configuración inicial es* $(q_0, \underline{B}wB, \underline{B}, \ldots, \underline{B})$ *y la configuración final de la sucesión es de la forma* $(q, \underline{B}wB, \underline{B}, \ldots, \underline{B})$*, con* $q = q_A$.

*De esta manera, una palabra* $w$ *es aceptada por* $M$ *en* $n$ *pasos si*

$$(q_0, \underline{B}wB, \underline{B}, \ldots, \underline{B}) \vdash_M^n (q_A, \underline{B}wB, \underline{B}, \ldots, \underline{B}).$$

### 2.1.2 *Máquinas de Turing multicinta no deterministas.*

Básicamente, una máquina de Turing multicinta no determinista será similar a la definida en 2.1.4 añadiendo la característica de que, a lo largo de una computación, de una configuración dada pueden derivarse más de una configuración. Se simula de esta forma la posibilidad de que una máquina pueda tener más de una acción posible partiendo de un determinado estado.

**Definición 2.1.10.** *Una máquina de Turing multicinta no determinista (en adelante* NTM*)* $M$ *queda definida por* $(Q, \Sigma, \Gamma, \Delta, q_0, k)$ *donde* $Q, \Sigma, \Gamma, q_0$ *y* $k$ *son similares a los dados en la definición 2.1.4, y* $\Delta$*, denominada relación de transición, será un subconjunto de:*

$$\left((Q - \{q_A, q_R\}) \times (\Sigma \cup \{B\}) \times (\Gamma \cup \{B\})^k\right) \times \left(Q \times (\Gamma \cup \{B\})^k \times \{-1, 0, 1\}^{k+1}\right)$$

La definición de configuración de una máquina de Turing determinista hecha en 2.1.5 sigue siendo válida en el caso de las NTM, ya que la diferencia sustancial se encuentra en la forma en que una configuración puede ser o no sucesora de otra.

**Definición 2.1.11.** *Si se consideran ahora las mismas condiciones dadas en los prolegómenos de la definición 2.1.7, entonces se dice que $\beta$ es una configuración sucesora de $\alpha$ (denotado por $\alpha \vdash_M \beta$), donde $\alpha$ y $\beta$ son configuraciones de la computación de una palábra $x$ en una* NTM *$M$, si:*

$$(q, a, \sigma_1, \ldots, \sigma_k, r, \tau_1, \ldots, \tau_k, m_0 - n_0, m_1 - n_1, \ldots, m_k - n_k) \in \Delta$$

Teniendo en cuenta este último cambio, las nociones de computación y aceptación en una NTM van a ser análogas a las de una máquina de Turing determinista. Hay que resaltar que para una configuración inicial $C_1$, una NTM puede producir más de una computación. De hecho las NTM producen un árbol de computación que tiene como raíz la configuración inicial $(q_0, \underline{B}xB, \underline{B}, \ldots, \underline{B})$. De cada nodo interno $\alpha$ se tendrá un conjunto de nodos hijos $\beta_0, \ldots, \beta_i$ donde $\alpha \vdash \beta_i$ para todo $i$. De esta forma, la sucesión de configuraciones que hacen que una palabra $w \in \Sigma^*$ sea aceptada se corresponderá con un camino en dicho árbol que lleva a una configuración cuyo estado es de aceptación, aunque puedan existir otros caminos que sean infinitos o que lleven a configuraciones cuyos estados sean $q_R$.

### 2.1.3 *Máquinas de Turing de Acceso Aleatorio.*

En los modelos computacionales dados hasta ahora, cualquier cálculo que dependa de todos los bits de la palabra de entrada tendrá al menos tiempo lineal sobre la longitud $n$ de la misma. Esto se debe al hecho de que recorrer la palabra de entrada al menos una vez consume $n$ pasos con total seguridad. Para conseguir tiempos sublineales se introduce el concepto de *acceso aleatorio* en dichos modelos, lo que significa que se requerirán dos cintas de trabajo más: una cinta de *consulta* que contendrá un índice en representación binaria y otra cinta de *respuesta* en cuya primera celda se puede almacenar el contenido de la celda de la cinta de entrada cuyo índice es el señalado en la cinta de consulta.

Tanto la definición como los elementos que caracterizan una máquina de Turing de Acceso Aleatorio (en adelante RATM) son similares a los dados para las máquinas de Turing, cambiando conceptualmente en lo siguiente:

- La cinta de entrada no tiene cabeza lectora.
- Se añade un nuevo estado $q_I$ en el que se escribe en la primera celda de la cinta de respuesta el bit de la palabra de entrada cuyo índice es el entero escrito en notación binaria en la cinta de consulta. En el caso en que dicho índice sea mayor que la longitud de la palabra de entrada se escribe el símbolo B en la cinta de consulta.
- El alfabeto $\Gamma$ utilizado en las cintas de trabajo deben contene los símbolos $\{0, 1\}$ utilizados en la cinta de consulta.
- Como es de esperar, la relación (o función) de transición $\Delta$ ($\delta$) cambia acorde al nuevo comportamiento de la máquina.

**Definición 2.1.12.** *Una máquina de Turing de Acceso Aleatorio* M *(en adelante* RATM*) queda definida por la* 6*–tupla* $(Q, \Sigma, \Gamma, \Delta, q_0, k)$ *donde* $k \in \mathbb{N}$ *y:*

- Q *es un conjunto finito de estados dentro del cual se encuentran:*
  - *Se denomina* $q_0$ *al estado inicial. Es el estado del que parte una máquina cada vez que esta se ejecuta sobre una palabra.*
  - $q_A$ *y* $q_R$ *representan al estado de aceptación y el estado de rechazo, respectivamente.*
  - $q_I$ *es el estado en el que se lee la celda de la cinta de entrada cuyo índice es indicado en la cinta de consulta y la respuesta se escribe en la primera celda de la cinta de respuesta.*
- $\Sigma$ *es un alfabeto finito que se usa como sólo lectura.* $\Sigma \cup \{B\}$ *contiene los símbolos que aparecerán en la cinta de entrada.*
- $\Gamma$ *es un alfabeto finito de lectura/escritura.* $\Gamma \cup \{B\}$ *será el alfabeto utilizado en las cintas de trabajo y, por tanto,* $\{0, 1\} \subseteq \Gamma$ *ya que la cinta de consulta contendrá un entero en forma binaria.*
- *En el caso de una* RANTM*, la relación de transición* $\Delta$ *será un subconjunto de:*

$$\left((Q - \{q_A, q_R\}) \times (\Gamma \cup \{B\})^{k+2}\right) \times \left(Q \times (\Gamma \cup \{B\})^{k+2} \times \{-1, 0, 1\}^{k+2}\right)$$

**Definición 2.1.13.** *Una* configuración de una RATM *será un elemento de* $Q \times (\Sigma \cup \{B\})^* \times (\Gamma \cup \{B\})^{*k+1} \times (\Sigma \cup \{B\}) \times \mathbb{N}^{k+1}$.

*Una configuración será de aceptación o de rechazo cuando sea de la forma* $\alpha = (q, \ldots)$ *donde* $q = q_A$ *ó* $q = q_R$ *(resp.).*

Se observa en esta definición que una configuración es la representación abstracta de una tupla en la que se indican respectivamente: estado actual de la máquina, contenido de la cinta de entrada, contenido de las cintas de trabajo y de consulta, contenido de la cinta de respuesta y posición de las cabezas de lectura de las cintas de trabajo y de consulta.

**Definición 2.1.14.** *Sean* $\alpha$ *y* $\beta$ *dos configuraciones de una* RATM M *de la forma:*

$$\begin{aligned} \alpha &= (q, BxB, \alpha_1, \ldots, \alpha_{k+1}, a, n_1, \ldots, n_{k+1}) \\ \beta &= (r, BxB, \beta_1, \ldots, \beta_{k+1}, b, n_1, \ldots, n_{k+1}) \end{aligned}$$

*donde para cada* $i \in \{1, \ldots, k+1\}$ *se tiene:*

*a)* $\alpha_i = u_i \sigma_i v_i$ *y* $\beta_i = u_i \tau_i w_i$ *con* $\sigma_i, \tau_i \in \Sigma \cup \{B\}$ *y* $u_i, v_i, w_i \in (\Sigma \cup \{B\})^*$ *(particularmente,* $\sigma_{k+1}, \tau_{k+1} \in \{0, 1, B\}$ *y* $u_i, v_i, w_i \in (\{0, 1, B\})^*$*).*

*b)* $|u_i| = n_i$.

*En estas condiciones,* $\beta$ *es la configuración* sucesora *de* $\alpha$ *(que expresamos* $\alpha \vdash_M \beta$*) sobre una palabra* x *si:*

$$(q, \sigma_1, \ldots, \sigma_{k+1}, a, r, \tau_1, \ldots, \tau_{k+1}, b, m_1 - n_1, \ldots, m_{k+1} - n_{k+1}) \in \Delta$$

*verificando para todo* $i \in \{1, \ldots, k+1\}$ *lo siguiente:*

1. *Si* $q = q_I$ *entonces* $\sigma_i = \tau_i$ *y* $m_i = n_i$ *y además* $b$ *será igual al bit de* $BxB$ *cuyo índice es el entero con representación binaria* $\alpha_{k+1}$. *Si dicho entero es mayor que* $|BxB|$ *entonces* $b = B$.

2. $m_i < |\beta_i|$.

3. *Se cumple una de la siguientes opciones:*
   - $v_i = w_i$.
   - $v_i = \varepsilon$, $w_i = B$ *y* $m_i = n_i + 1$.

### 2.1.4 *Máquinas de Turing Alternantes.*

Una máquina de Turing Alternante (en adelante ATM) puede verse como un modelo computacional basado en una RATM en la que el conjunto de estados no finales (i.e., que no son de aceptación o rechazo) está subdividido en estados *universales* ($\wedge$) y *existenciales* ($\vee$). Este modelo será el utilizado para dar las definiciones de clases de complejidad en tiempo logarítmico y polinomial que se usarán a lo largo del presente trabajo.

**Definición 2.1.15.** *Una máquina de Turing Alternante (en adelante* ATM*)* $M$ *queda definida por la tupla* $(Q, \Sigma, \Gamma, \Delta, q_0, k, \ell)$ *donde* $Q, \Sigma, \Gamma, \Delta, q_0, k, q_A, q_R$ *se dan de la misma forma que en la definición 2.1.3 (*RATM*) mientras que* $\ell$ *es una función total tal que* $\ell : (Q - \{q_A, q_R\}) \to \{\wedge, \vee, q_A, q_R\}$.

Las nociones de *configuración* y *computación* de una ATM vienen directamente heredadas de las máquinas de Turing no deterministas que fueron desarrolladas en el apartado 2.1.2. De esta forma, el cálculo de una ATM sobre una palabra $x$ tendrá forma de árbol en el que cada nodo no terminal es una configuración cuyo estado es existencial o universal. Este detalle va a permitir añadir restricciones a cuando una palabra es aceptada por una ATM.

**Definición 2.1.16.** *Una* computación de aceptación, $T$, *es un subárbol del árbol de computación de una* ATM $M$ *sobre una palabra* $x$, *en el que, para cada configuración* $\alpha \in T$ *con estado* $q$, *se tiene:*

- *La raíz de* $T$ *es la configuración inicial de* $M$ *sobre* $x$.
- *Si* $\alpha$ *es una hoja de* $T$, *entonces* $\alpha$ *es una configuración de aceptación (es decir, su estado es* $q_A$*).*
- *Si* $\alpha$ *es una* configuración universal *(*$\ell(q) = \wedge$*) entonces todos los hijos de esta configuración pertenecen al subárbol* $T$. *Dicho de otra forma, para todo* $\beta$ *tal que* $\alpha \vdash_M \beta$ *se tiene que* $\beta \in T$.
- *Si* $\alpha$ *es una* configuración existencial *(*$\ell(q) = \vee$*) entonces existe al menos un hijo de esta configuración que pertenece a* $T$, *es decir,* $\exists \beta \in T$ *tal que* $\alpha \vdash_M \beta$.

# 3

# COMPLEJIDAD COMPUTACIONAL Y ÁLGEBRAS DE FUNCIONES.

Antes de seguir progresando hacia el objetivo de este trabajo, va a ser necesario anticipar una definición concreta tanto de la noción de álgebra de funciones como de clase de complejidad y, dentro de éstas últimas, destacar algunas de ellas que serán materia de estudio en capítulo posteriores.

## 3.1 COMPLEJIDAD COMPUTACIONAL.

En un primer acercamiento a la teoría de la Complejidad, sería razonable describirla como el estudio de la dificultad para resolver problemas computacionalmente en términos de la cantidad de recursos, normalmente tiempo y espacio (memoria), que consume la solución de dichos problemas en un modelo de computación. Para ello, lejos de interesarse por el análisis de algoritmos puntuales, la teoría de la Complejidad pretende estudiar clases de complejidad que puedan aunar problemas calculables con cantidades similares de estos recursos.

En lo que sigue será importante recordar el concepto de orden de una función:

**Definición 3.1.1.** *Sean* $f$ *y* $g$ *funciones de* $\mathbb{N}$ *en* $\mathbb{R}_{\geq 0}$. *Diremos que* $f$ *es del orden de* $g$, *notado* $f = \mathcal{O}(g(n))$, *si existen una constante positiva* $c$ *y un* $n_0 \in \mathbb{N}$ *tal que para todo* $n \geq n_0$ *se cumple que* $f(n) \leq c \cdot g(n)$. *Es decir, llamamos* orden *de* $g$ *a:*

$$\mathcal{O}(g) = \{h : (\exists c > 0)(\exists n_0)(\forall n \geq n_0)[h(n) \leq c \cdot g(n)]\}.$$

Los siguientes apartados tratan de describir las clases de complejidad que pueden definirse utilizando los modelos computacionales descritos en el capítulo anterior.

### 3.1.1 *Complejidad en máquinas de Turing deterministas.*

El número de pasos (*tiempo*) y de celdas usadas (*espacio*) por una máquina de Turing refleja la complejidad de los cálculos que realiza dicha máquina. Esto induce a clasificar los lenguajes aceptados por una máquina de Turing dependiendo del consumo que ésta hace de ambas variables durante una computación.

**Definición 3.1.2.** *Sea* $T : \mathbb{N} \to \mathbb{R}_{\geq}0$, *se dice que una máquina de Turing determinista* $M$ acepta *un lenguaje* $L \subseteq \Sigma^*$ *en tiempo* $T(n)$ *si* $L$ *es aceptado por* $M$ *y para cada palabra* $w \in L(M)$ *de longitud* $n$, $w$ *es aceptada en a los sumo* $T(n)$ *pasos.*

*En las mismas condiciones, dada* $S : \mathbb{N} \to \mathbb{R}_{\geq}0$, $M$ *acepta* $L$ *en un espacio* $S(n)$ *si para cada configuración,* $C = (q, BwB, \alpha_1, \ldots, \alpha_k, n_0, \ldots, n_k)$, *de una computación de aceptación de* $M$ *para una palabra* $w \in L(M)$, *con* $|w| = n$, *se tiene:*

$$\sum_{j=1}^{k} |\alpha_j| \leq S(n).$$

**Definición 3.1.3.** *Un lenguaje* $L \subseteq \Sigma^*$ *es* decidible *por una máquina de Turing determinista* $M$ *en tiempo* $T(n)$ *si* $L$ *(resp.* $\Sigma^* - L$*) es la colección de palabras sobre las cuales* $M$ *para en el estado de aceptación* $q_A$ *(resp.* $q_R$*) y además, sobre toda palabra* $w \in \Sigma^*$ *de longitud* $n$, $M$ *para en a lo sumo* $T(n)$ *pasos.*

*Como en la definición anterior, considerando las mismas condiciones, se puede decir que* $L$ *es decidible por* $M$ *en espacio* $S(n)$ *si en la computación de toda palabra* $w$ *el número máximo de celdas visitadas en las cintas de trabajo de* $M$ *es* $S(n)$.

Aunque en un principio las nociones de aceptación y decidibilidad sean conceptos diferentes, a lo largo del trabajo se emplearán de manera indistinta. Este hecho se justifica en que para la mayoría de clases de complejidad que serán consideradas, las máquinas implicadas pueden modificarse de forma que rechacen una palabra en caso de que ésta no sea aceptada.

La siguiente definición permite clasificar los lenguajes $L \subseteq \Sigma^*$ aceptados por alguna máquina de Turing determinista:

**Definición 3.1.4.** *Sean* $T, S$ *funciones de* $\mathbb{N}$ *en* $\mathbb{R}_{\geq 0}$. *Se definen las siguientes colecciones de lenguajes, que llamamos* clases de complejidad, *como:*

$$\begin{aligned}
\textsc{dtime}(T(n)) &= \{L \subseteq \Sigma^* : L \textit{ es aceptado por una } \textsc{tm} \\
&\quad \textit{en tiempo } \mathcal{O}(T(n))\} \\
\textsc{dspace}(S(n)) &= \{L \subseteq \Sigma^* : L \textit{ es aceptado por una } \textsc{tm} \\
&\quad \textit{en espacio } \mathcal{O}(S(n))\} \\
\textsc{DTimeSpace}(T(n), S(n)) &= \{L \subseteq \Sigma^* \textit{ tal que } L \textit{ es aceptado por una } \textsc{tm} \\
&\quad \textit{en tiempo } \mathcal{O}(T(n)) \textit{ y espacio } \mathcal{O}(S(n))\}
\end{aligned}$$

Con esta definición podemos señalar algunas clases de complejidad especialmente relevantes:

**Definición 3.1.5.** *Definimos los siguientes conjuntos:*

$$\begin{aligned}
\textsc{ptime} &= \bigcup_{k \geq 1} \textsc{dtime}(n^k) \\
\textsc{pspace} &= \bigcup_{k \geq 1} \textsc{dspace}(n^k) \\
\textsc{etime} &= \bigcup_{k \geq 1} \textsc{dtime}(2^{k \cdot n}) \\
\textsc{exptime} &= \bigcup_{k \geq 1} \textsc{dtime}(2^{n^k})
\end{aligned}$$

### 3.1.2 *Complejidad en máquinas de Turing no deterministas.*

Es claro que también se pueden definir conceptos sobre el *tiempo* y *espacio* requeridos en la computación de una máquina de Turing multicinta no determinista.

**Definición 3.1.6.** *Una* NTM $M$ decide *una palabra* $w \in \Sigma^*$ *de longitud* $n$ *en tiempo* $T(n)$ *si la profundidad del árbol de computación asociado es a lo sumo* $T(n)$.

*Igualmente, se dice que* $w$ *es aceptada por* $M$ *en espacio* $S(n)$ *si para cada configuración* $\alpha$ *del árbol de computación asociado, el número de celdas usado las cintas de trabajo es a lo sumo* $S(n)$.

Cabe recordar de nuevo que tanto para los modelos de computación como para las clases de complejidad tratadas en este trabajo, se identificarán los conceptos de aceptación y decidibilidad de una palabra por lo motivos ya mencionados en el apartado anterior.

De nuevo se pueden señalar algunas colecciones de lenguajes dependiendo de las nociones de tiempo y espacio dadas en la última definición.

**Definición 3.1.7.** *Sean* $T, S$ *funciones de* $\mathbb{N}$ *en* $\mathbb{R}_{\geq 0}$. *Se definen las siguientes clases de complejidad:*

$$\begin{aligned}
\text{NTIME}(T(n)) &= \{L \subseteq \Sigma^* : L \text{ es aceptado por una NTM} \\
&\quad \text{en tiempo } \mathcal{O}(T(n))\} \\
\text{NSPACE}(S(n)) &= \{L \subseteq \Sigma^* : L \text{ es aceptado por una NTM} \\
&\quad \text{en espacio } \mathcal{O}(S(n))\} \\
\text{NTIMESPACE}(T(n), S(n)) &= \{L \subseteq \Sigma^* : L \text{ es aceptado por una NTM} \\
&\quad \text{en tiempo } \mathcal{O}(T(n)) \text{ y espacio } \mathcal{O}(S(n))\}
\end{aligned}$$

Especialmente importante resulta en este modelo la clase de complejidad $\text{NP} = \text{NTIME}(n^{\mathcal{O}(1)})$, es decir, el conjunto de problemas que pueden resolverse en tiempo polinómico por una máquina de Turing no determinista.

Hay que señalar que tanto el concepto de *computación* como el de *aceptación de un lenguaje* $L$ de una máquina de Turing $M$ permanecen inalterados si añadimos a esta la capacidad de acceso aleatorio descrita en 2.1.3.

### 3.1.3 *Complejidad en máquinas de Turing alternantes.*

De manera similar a los modelos anteriores vamos a definir cuándo una máquina de Turing alternante acepta un cierto lenguaje $L(M)$ para, posteriormente, dar algunas clases de complejidad especialmente interesantes.

**Definición 3.1.8.** *Decimos que una* ATM $M$ *acepta una palabra* $x$ *si existe una computación de aceptación de* $M$ *sobre* $x$. *En cualquier otro caso, se dice que la palabra* $x$ *es rechazada.*

**Definición 3.1.9.** *Una* ATM $M$ acepta *un lenguaje* $L(M)$ *en tiempo* $T(n)$ *si para cada palabra* $w \in L(M)$ *de longitud* $n$ *existe una computación de aceptación* $T$ *con, a lo sumo, una profundidad* $T(n)$.

M *acepta* $L(M)$ *en espacio* $S(n)$ *si a lo sumo se utilizan* $S(n)$ *celdas por las cintas de trabajo en cualquier nodo del subárbol* T.

Además del tiempo y el espacio habrán de ser tenidas en cuenta las veces que en un árbol de aceptación se cambia de estados existenciales a universales (o viceversa).

**Definición 3.1.10.** *El número de* alternancias *que una* ATM M *hace en la computación de un árbol de aceptación* T *vendrá dado por el número máximo de veces que se cambian entre nodos existenciales y universales en un camino desde la raíz hasta una hoja de* T.

Tenemos entonces que una máquina de Turing no determinista es un caso particular de las alternantes en el que todos los estados son de tipo existencial.

**Definición 3.1.11.** *Se dan las siguientes colecciones de lenguajes:*

$$\begin{aligned} \text{ATIME}(T(n)) &= \{L \subseteq \Sigma^* : L \text{ es aceptado por una ATM en tiempo } \mathcal{O}(T(n))\} \\ \text{ASPACE}(S(n)) &= \{L \subseteq \Sigma^* : L \text{ es aceptado por una ATM en espacio } \mathcal{O}(S(n))\} \end{aligned}$$

Como clases a tener en cuenta cabe señalar las siguientes:

$$\begin{aligned} \text{ALOGTIME} &= \text{ATIME}(\mathcal{O}(\log n)) \\ \text{APOLYLOGTIME} &= \bigcup_{k\geq 1} \text{ATIME}(\mathcal{O}(\log^k n)) \\ \text{ALINTIME} &= \text{ATIME}(\mathcal{O}(n)) \end{aligned}$$

**Definición 3.1.12.** *Llamamos* $\Sigma_k - \text{TIME}(T(n))$ *a la colección de lenguajes aceptados por una* ATM *en tiempo* $\mathcal{O}(T(n))$ *donde cada computación de aceptación tiene, a lo sumo,* k *alternancias y comienza siempre en un estado existencial.*

Con esto podemos definir las siguientes jerarquías que vamos a notar como:

- LH = $\bigcup \Sigma_k - \text{TIME}(\mathcal{O}(\log(n)))$, jerarquía de tiempo logarítmico.
- LTH = $\bigcup \Sigma_k - \text{TIME}(\mathcal{O}(n))$, jerarquía de tiempo lineal.
- PH = $\bigcup \Sigma_k - \text{TIME}(n^{\mathcal{O}(1)})$, jerarquía de tiempo polinomial.

## 3.2 ÁLGEBRA DE FUNCIONES.

Las álgebras de funciones son familias de funciones generadas mediante la aplicación una serie de operadores a unas funciones base dadas.

**Definición 3.2.1.** *Sean:*

- $\mathcal{X}$ *un conjunto de funciones, y*
- OP *una colección de* operadores *(es decir, aplicaciones que tienen como argumento una o más funciones y devuelven una función como valor).*

*Se dice que un* álgebra de funciones $[\mathcal{X}; \text{OP}]$ *es el menor conjunto de funciones que contienen a* $\mathcal{X}$ *y es cerrado bajo las operaciones de* OP. *Los representantes de un álgebra de funciones pueden ser dados aplicando inducción sobre las funciones de* $[\mathcal{X}; \text{OP}]$.

Un ejemplo de álgebra de funciones, que aparecerá más adelante, puede ser el conjunto de funciones primitivas recursivas, definido como $[0, I, s; \text{COMP}, \text{PR}]$. Tenemos entonces que sus funciones iniciales son la función constante $0$, las proyecciones $I_i^n(x_1, \ldots, x_n)$ y la función siguiente $s$, que veremos que calcula $s(x) = x + 1$. Como operadores del álgebra se tienen la composición, COMP y la recursión primitiva, PR.

Dentro de un álgebra de funciones, es intuitivo caracterizar predicados, de forma que podemos trabajar con ellos tratándolos como una función más del álgebra:

**Definición 3.2.2.** *Llamaremos* función característica $c_P(\vec{x})$ *de un predicado* $P$ *a:*

$$c_P(\vec{x}) = \begin{cases} 1 & \textit{si } P(\vec{x}) \\ 0 & \textit{en caso contrario.} \end{cases}$$

*Como es usual, frecuentemente identificaremos* $P$ *y* $c_P$. *Además, si* $\mathcal{F}$ *es una clase de funciones, entonces denotamos por que* $\mathcal{F}_*$ *a la clase de predicados cuya función característica pertenece a* $\mathcal{F}$.

Hasta ahora, los modelos computacionales que hemos introducido tienen únicamente la capacidad de aceptar, o no, un lenguaje dado, aunque, desde un principio, nos hemos referido a ellos como herramientas capaces de resolver problemas computacionalmente. Por una parte, es posible hacer que reconozcan predicados $R \subseteq (\Sigma)^k$ si permitimos que éstos puedan tener entradas de la forma:

$$Bx_1Bx_2B \ldots Bx_nB,$$

es decir, la entrada de $n$ palabras $x_i \in \Sigma^*$, cada una de ellas separada de la otra por el carácter $B$. Si, además de esto, añadimos a la máquina una cinta de salida, con una cabeza únicamente capaz de escribir y moverse a la derecha, tendremos que las máquinas de Turing pueden calcular funciones de aridad $n$. Las siguientes definiciones vienen a formalizar estas nociones, aunque de un modo algo distinto que nos permitirá incluir en nuestra definición funciones computables en tiempo sublineal.

**Definición 3.2.3.** *Una función* $f : \mathbb{N}^n \to \mathbb{N}$ *tiene un* crecimiento polinomial *si:*

$$|f(x_1, \ldots, x_n)| = \mathcal{O}(\max_{1 \leq i \leq n} |x_i|^k), \textit{ para algún } k.$$

*Igualmente, diremos que* $f(x_1, \ldots, x_n)$ *tiene un* crecimiento lineal *si:*

$$|f(x_1, \ldots, x_n)| = \mathcal{O}(\max_{1 \leq i \leq n} |x_i|).$$

**Definición 3.2.4.** *Definimos el* grafo *de* $f(\vec{x})$ *como el predicado* $G_f(\vec{x}, y)$ *que se satisface si y sólo si* $f(\vec{x}) = y$. *De igual manera, definimos el* bitgrafo $B_f$ *de modo que* $B_f(\vec{x}, i)$ *se satisface si y sólo si el* $i$*-ésimo bit de* $f(\vec{x})$ *es* $1$.

**Definición 3.2.5.** *Si $\mathcal{C}$ es una clase de complejidad, entonces definimos $\mathcal{FC}$ como la clase de funciones de crecimiento polinomial cuyo bitgrafo pertenece a $\mathcal{C}$. También se define $\mathtt{Lin}\mathcal{FC}$ (que abreviaremos como $\mathcal{GC}$) como la clase de funciones de crecimiento lineal cuyo bitgrafo está en $\mathcal{C}$.*

Estas nociones permiten, por ejemplo, observar que un modelo computacional que opera en tiempos sublineales (RATM), contrariamente a lo que se podría pensar, tienen una capacidad de cómputo relevante:

**Proposición 3.2.6.** *Existe una función binaria biyectiva decodificable por una* RATM *en tiempo logarítmico.*

**Demostración.** Para dar un esquema de la prueba supongamos que $M$ es una RATM que calcula el bitgrafo del $i$-ésimo bit de la codificación $\langle x, y\rangle$, donde $\langle x, y\rangle : \Sigma^* \times \Sigma^* \to \Sigma^*$ está definida como:

$$z = \langle x, y\rangle = \sigma(|x|)11\sigma(|y|)11xy$$

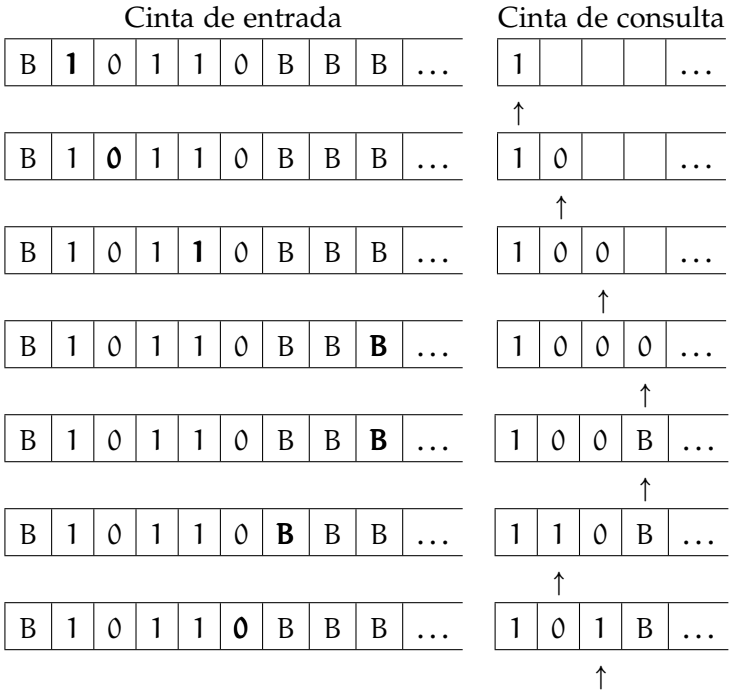
con $\sigma : \Sigma^* \to \Sigma^*$, que reemplaza cada $1$ de la palabra de entrada por $01$ y cada $0$ por $00$. De esta forma, para decodificar $|x|$, la máquina $M$ sólo tiene que recorrer el principio de la entrada $z$ hasta encontrar dos unos seguidos. Si cada vez que avanza dos celdas, va anotando este hecho en una cinta de trabajo, $M$ decodificará $|x|$ en $2 \cdot ||x|| + 2$ pasos, es decir, en un tiempo menor que $\log |z|$. De forma similar se decodifica $|y|$, esta vez comenzando el proceso en el índice $2 \cdot ||x|| + 3$.

Encontrar el $i$-ésimo bit de $x$ (ó $y$) requerirá calcular el número entero $2 \cdot ||x|| + 2 + 2 \cdot ||y|| + 2 + i$ (ó $2 \cdot ||x|| + 2 + 2 \cdot ||y|| + 2 + |x| + i$) y escribirlo en la cinta de consulta, teniendo en cuenta que el entero $2 \cdot ||x|| + 2 + 2 \cdot ||y||$ se puede obtener de $|z| \dot{-} (|x| + |y| + 2)$. Así, para que $M$ realice este cálculo en tiempo logarítmico debe ser capaz de sumar y restar enteros menores que la longitud de $\log |z|$ y calcular la longitud de la entrada en dicho tiempo.

Por una parte, sumar dos enteros en una máquina de Turing va a suponer recorrer todos los bits de ambos números y hacer la suma de cada par de bits uno a uno. Esto tiene orden lineal sobre la longitud de los enteros, pero como en las sumas en las que estamos interesados dichos enteros tienen longitud menor que $\log |x|$, se tiene que la suma de estos será de orden sublogarítmico.

Si atendemos ahora al cálculo de $|z| = n$, tan sólo tenemos que buscar el menor $i$ tal que $n < 2^i$. Si lo hacemos secuencialmente comenzando en $i = 2$ no es difícil ver que este proceso tiene tiempo $\mathcal{O}(\log n)$. Una vez encontrado tendremos a $n$ en el intervalo $[2^{i-1}, 2^i]$, por lo que podremos hacer una búsqueda binaria que nos llevará a $n$ en tiempo $\mathcal{O}(\log n)$. Este procedimiento puede realizarse en una RATM utilizando para ello la cinta de consulta, de forma que al final del cálculo la longitud de la entrada quede reflejada en la misma.

Veamos un ejemplo de esto último, observando la evolución de la cinta de consulta. En la cinta de entrada se introduce la palabra 10110 de longitud 5:




Con lo que en la cinta de consulta queda 101, que es la representación binaria de 5, es decir, la longitud de la palabra de entrada. □

# 4

# JERARQUÍA DE TIEMPO LOGARÍTMICO LH.

A lo largo de este capítulo, en base a los resultados de Peter Clote en [3], vamos a dar y describir detalladamente un álgebra de funciones con el que se pueda interpretar $\mathcal{F}_{\text{LH}}$, es decir, las funciones en tiempo logarítmico.

La idea de caracterizar clases de complejidad pequeñas, como la que nos ocupa, se basa en resultados anteriores concernientes a clases de complejidad mayores. Ejemplo de ello son las álgebras de Cobham y Ritchie dadas para PTIME y LINSPACE, respectivamente, que son presentadas en el siguiente capítulo de este trabajo.

Tal y como se esbozó en la introducción de las máquinas de Turing de Acceso Aleatorio, para conseguir un álgebra que se corresponda con $\mathcal{F}_{\text{LH}}$ será necesario trabajar con la representación binaria de números enteros.

**Definición 4.0.1.** *Definimos las siguientes funciones:*

- *Función* $0$*:* $0(x) = 0$.
- *Función* sucesor*:* $s(x) = x + 1$.
- *Funciones* sucesores binarios*:* $s_0(x) = 2 \cdot x$ *y* $s_1(x) = 2 \cdot x + 1$.
- *Función* proyección $n$*-ésima:* $I^n_k(x_1, \ldots, x_n) = x_k$. *Notaremos por* $I$ *al conjunto de todas las funciones proyección.*

**Definición 4.0.2.** *Decimos que la función* $f : \mathbb{N}^n \to \mathbb{N}$ *esta definida por* composición *(*COMP*) de las funciones* $h, g_1, \ldots, g_m$ *(de aridad adecuada) si:*

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)).$$

*La función* $f : \mathbb{N}^n \to \mathbb{N}$ *estará definida por* recursión primitiva *(*PR*) a partir de las funciones* $g, h$ *si*

$$\begin{aligned} f(0, \vec{y}) &= g(\vec{y}) \\ f(x+1, \vec{y}) &= h(x, \vec{y}, f(x, \vec{y})) \end{aligned}$$

*siendo* $g$ *una constante si* $n = 0$.
*Al álgebra de funciones* $[0, I, s; \text{COMP}, \text{PR}]$ *lo llamamos* conjunto de funciones primitivas recursivas*, notándolo por* $\mathcal{PR}$*. Llamaremos* $\mathcal{PR}_1$ *a la colección de funciones primitivas recursivas* $1$*–árias.*
*Una función* $f$ *estará definida por* iteración *(*ITER*) si existe una función* $g$ *tal que:*

$$\begin{aligned} f(0) &= 0 \\ f(x+1) &= g(f(x)). \end{aligned}$$

*Para finalizar damos la operación* suma *(*ADD*) simplemente como* $\text{ADD}(f, g)(x) = f(x) + g(x)$.

Debido a que la recursión primitiva define $f(x+1)$ en términos de $f(x)$, lo que hace que calcular $f(x)$ requiera aproximadamente $2^{|x|}$ pasos (es decir, orden exponencial), será necesario buscar un operador de recursión diferente capaz de caracterizar clases de complejidad en tiempo sublineal. Un recurso que posibilitará la definición de clases de funciones de menor complejidad son los esquemas de recursión sobre la notación.

**Definición 4.0.3.** *Sean* $g, h_0, h_1$ *funciones tales que* $h_0(x,\vec{y}), h_1(x,\vec{y}) \leq 1$. *Una función* $f$ *esta definida mediante* recursión por concatenación sobre notación *(CRN) si:*

$$\left.\begin{array}{l} f(0,\vec{y}) = g(\vec{y}) \\ f(s_0(x),\vec{y}) = s_{h_0(x,\vec{y})}(f(x,\vec{y})), \\ f(s_1(x),\vec{y}) = s_{h_1(x,\vec{y})}(f(x,\vec{y})) \end{array}\right\} \textit{si } x \neq 0.$$

Continuando con esta idea, se dan ahora una serie de funciones sobre enteros orientadas al trabajo sobre la representación binaria de los mismos.

**Definición 4.0.4.** *Definimos las siguientes funciones:*

- *La longitud de* $x$: $|x| = \lceil \log(x+1) \rceil$. *Será habitual en lo que sigue usar también* $||x|| = |(|x|)|$.
- *Función* MOD2: $\text{MOD2}(x) = x - 2 \cdot \lfloor \frac{x}{2} \rfloor$
- *Función* BIT: $\text{BIT}(i,x) = \text{MOD2}(\lfloor \frac{x}{2^i} \rfloor)$.
- *Función* smash: $x\#y = 2^{|x|\cdot|y|}$.

*Donde por* $\lfloor x \rfloor : \mathbb{R}_{\geq 0} \to \mathbb{N}$ *nos referimos a la función suelo, que calcula el mayor entero menor o igual que* $x$. *De forma similar,* $\lceil x \rceil$ *calcula el menor entero mayor o igual que* $x$.

Como es de esperar, si $x$ es cualquier entero, tendremos que $\text{MOD2}(x)$ dará último dígito por la derecha de $x$ en notación binaria y $\text{BIT}(i,x)$ devuelve el $i$–ésimo bit de $x$, también en binario.

Con las funciones y operadores dados hasta ahora se puede introducir el álgebra $A_0$.

**Definición 4.0.5.** *Llamamos* $A_0$ *al álgebra de funciones:*

$$[0, I, s_0, s_1, \text{BIT}, |x|, \#; \text{COMP}, \text{CRN}].$$

$A_0$ será el álgebra de funciones protagonista de esta sección y, de hecho, la que será asimilable a la jerarquía de tiempo logarítmico. Una descripción de la misma ayudará a obtener una codificación de cualquier computación de una ATM en tiempo logarítmico.

## 4.1 ELEMENTOS Y PROPIEDADES DE $A_0$.

Antes de comenzar con esta descripción es necesario señalar que las funciones constantes están en $A_0$. Por ejemplo, si se toma la función $f(x) = 5$, ésta se puede definir en $A_0$ como $f(x) = s_1(s_0(s_1(0)))$.

### 4.1.1 *Funciones reverso y signo.*

La función *reverso auxiliar*, $rev0$, se determina mediante CRN siguiendo:

$$\begin{cases} rev0(x,0) = 0 \\ rev0(x,s_i(y)) = s_{BIT(|y|,x)}(rev0(x,y)), \quad \text{con } i = 0,1. \end{cases}$$

Básicamente, $rev0$ devuelve los últimos (los menos significativos) $|y|$ bits de $x$ dados en sentido inverso.

Para todo $x \in \mathbb{N}$ se define la función *reverso* como:

$$rev(x) = rev0(x,x).$$

Es fácil concluir que $rev(x)$ devuelve el número natural cuya notación binaria es justamente la inversa $x$.

Dado que la recursión por concatenación sobre notación (CRN) es utilizada con asiduidad, será conveniente dar un ejemplo de la misma usando la función reverso como muestra. El cálculo detallado de $rev(6)$, cuya notación binaria es 110, viene dado por:

$$\begin{aligned} rev(6) &= rev(110) \\ &= rev0(110,110) \\ &= s_{BIT(|11|,110)}(rev0(110,11)) \\ &= s_1(s_{BIT(|1|,110)}(rev0(110,1))) \\ &= s_1(s_1(s_{BIT(0,110)}(rev0(110,0)))) \\ &= s_1(s_1(s_0(0))) \\ &= 3. \end{aligned}$$

Tenemos que $rev(6) = 3$, que en notación binaria es 11, justamente el reverso de 110.

Para dar la función signo de Kleene y su inversa, respectivamente $sg(x) = \text{mín}(x,1)$ y $\overline{sg}(x) = 1 - sg(x)$, se recurre a las funciones auxiliares $ones(x)$ y $pad(x)$:

- La función $ones(x)$ calcula el número cuya representación binaria consta únicamente de 1 y tiene una longitud $|x|$ o, lo que es lo mismo, $ones(x) = 2^{|x|} - 1$. Dentro de $A_0$ se puede calcular $ones(x)$ usando CRN:

$$\begin{cases} ones(0) = 0 \\ ones(s_i(x)) = s_1(ones(x)) \end{cases}$$

- Sea ahora la función $pad$ tal que $pad(x,y) = 2^{|y|} \cdot x$. De esta forma, $pad$ devuelve un natural cuya representación binaria coincide con la de $x$ añadiéndole a la derecha de la misma un número $|y|$ de ceros. Usando CRN se define como:

$$\begin{cases} pad(x,0) = x \\ pad(x,s_i(y)) = s_0(pad(x,y)). \end{cases}$$

Se definen en $A_0$ las funciones signo de Kleene como:

$$sg(x) = \text{BIT}(0, ones(x))$$
$$\overline{sg}(x) = \text{BIT}(0, pad(1, x)).$$

### 4.1.2 *Condicional.*

Se requiere un razonamiento algo más extenso para definir la función condicional en $A_0$. Se persigue obtener la función:

$$cond(x, y, z) = \begin{cases} y, & \text{si } x = 0 \\ z, & \text{en caso contrario.} \end{cases}$$

Para dar una representación de $cond$ en $A_0$ previamente se van a definir la función concatenación $*$ y las funciones auxiliares $cond0$, $cond1$ y $cond2$:

- La función concatenación, notada por $x * y$, calcula el número cuya representación binaria es la representación binaria de $x$ seguida de la de $y$. Puede definirse como:

$$\begin{cases} x * 0 = x \\ x * s_i(y) = s_i(x * y). \end{cases}$$

- La función $cond0(x, y)$ realiza el siguiente cálculo: si $x$ o el bit menos significativo de $y$ en notación binaria es 0, entonces $cond0$ devuelve 0, en caso contrario devolverá el natural cuya representación binaria sólo contiene 1 y su longitud es $|x|$, o sea:

$$cond0(x, y) = \begin{cases} 0, & \text{si } \text{BIT}(0, y) = 0 \\ 2^{|x|} - 1, & \text{en cualquier otro caso.} \end{cases}$$

  Se define $cond0$ usando CRN como:

$$\begin{cases} cond0(0, y) = 0 \\ cond0(s_i(x), y) = s_{\text{BIT}(0,y)}(cond0(x, y)) \end{cases}$$

- La función $cond1(x, y)$ será igual a cero siempre que $x$ o el bit más a la derecha de la notación binaria de $y$ sea 0, en caso contrario $cond1(x, y) = 1$, es decir:

$$cond1(x, y) = \begin{cases} 0, & \text{si } x = 0 \\ \text{BIT}(0, y), & \text{en cualquier otro caso.} \end{cases}$$

  Se define $cond1$ en $A_0$ como:

$$cond1(x, y) = sg(cond0(x, y)).$$

- Por último, se expone $cond2(x,y)$, que devuelve $y$ en caso de que $x \neq 0$:

$$cond2(x,y) = \begin{cases} 0, & \text{si } x = 0 \\ y, & \text{en cualquier otro caso.} \end{cases}$$

Usando CRN y la función $cond1$ se tiene:

$$\begin{cases} cond2(x,0) = 0 \\ cond2(x,s_i(y)) = s_{cond1(x,s_i(y))}(cond2(x,y)) \end{cases}$$

La función condicional $cond$ quedará entonces definida en $A_0$ como:

$$cond(x,y,z) = cond2(\overline{sg}(x),y) * cond2(x,z),$$

En el caso de $cond(x,y,z)$, la función concatenar opera en realidad como una disyunción, si el $sg(x) = 0$ entonces el resultado será $cond2(\overline{sg}(x),y)$ y, en caso contrario, $cond2(x,z)$.

La función $cond$ va a permitir definir predicados aplicando las operaciones booleanas AND, OR, NOT a otros predicados. De esta forma, si $P(\vec{x})$ y $Q(\vec{x})$ son predicados en $A_0$, se definen en $A_0$:

- $c_{\neg P}(\vec{x}) = \overline{sg}(c_P(\vec{x}))$.
- $c_{P\vee Q}(\vec{x}) = sg(c_P(\vec{x}) * c_Q(\vec{x}))$.
- $c_{P\wedge Q}(\vec{x}) = cond(\overline{sg}(c_P(\vec{x})), c_Q(\vec{x}), 0)$.

Además, se obtiene la definición por casos como una generalización de la función condicional:

$$f(\vec{x}) = \begin{cases} g_1(\vec{x}), & \text{si } P_1(\vec{x}) \\ g_2(\vec{x}), & \text{si } P_2(\vec{x}) \\ \qquad\vdots & \\ g_n(\vec{x}), & \text{si } P_n(\vec{x}), \end{cases}$$

donde los predicados $P_1, \ldots, P_n$ son disjuntos. Para ello basta con agrupar dos a dos los casos mediante la función condicional.

### 4.1.3 *Cuantificación fuertemente acotada.*

Como consecuencia directa de la función condicional se obtiene la cuantificación fuertemente acotada, la cual, siendo $P$ un predicado $(n+1)$-ario sobre $\mathbb{N}$, viene dada por:

- Cuantificación existencial fuertemente acotada:

$$(\exists x \leq |y|)P(x,\vec{z}) = \begin{cases} 1, & \text{si existe } x_0 \leq |y| \text{ tal que } P(x_0,\vec{z}) \\ 0, & \text{en caso contrario.} \end{cases}$$

- Cuantificación universal fuertemente acotada:

$$(\forall x \leq |y|)P(x,\vec{z}) = \begin{cases} 1, & \text{si para todo } x_0 \leq |y| \text{ se tiene } P(x,\vec{z}) \\ 0, & \text{en caso contrario.} \end{cases}$$

**Lema 4.1.1.** $(A_0)_*$ *es cerrada para los cuantificadores fuertemente acotados.*

**Demostración.** Sea $R(x,\vec{z})$ un predicado de $A_0$ (es decir, $c_R(x,\vec{z}) \in A_0$) y definimos $P(y,\vec{z}) = (\exists x \leq |y|)R(x,\vec{z})$. Utilizando CRN para dar la función:

$$\begin{cases} f(0,\vec{z}) = 0 \\ f(s_i(x),\vec{z}) = s_{R(|x|,\vec{z})}(f(x,\vec{z})) \end{cases}$$

entonces $P(y,\vec{z}) = sg(f(s_1(y),\vec{z}))$ estará en $A_0$.
Para el cuantificador universal, se supone $Q(y,\vec{z}) = (\forall x \leq |y|)R(x,\vec{z})$ y, de forma similar, se obtiene la función:

$$\begin{cases} g(0,\vec{z}) = 0 \\ g(s_i(x),\vec{z}) = s_{\overline{sg}(R(|x|,\vec{z}))}f(x,\vec{z}) \end{cases}$$

con lo que $Q(y,\vec{z}) = \overline{sg}(g(s_1(y),\vec{z}))$ y así $Q(y,\vec{z}) \in A_0$.

### 4.1.4 *Minimización y maximización fuertemente acotadas.*

De forma similar al lema 4.1.1 se puede observar que $A_0$ también es cerrada bajo minimización y maximización fuertemente acotadas.

**Definición 4.1.2.** *Una función* $f$ *está definida por minimización fuertemente acotada (*SBMIN*) a partir de* $g$ *si:*

$$f(x,\vec{y}) = \begin{cases} \text{mín}\{i \leq |x| : g(i,\vec{y}) = 0\}, & \textit{si existe,} \\ 0, & \textit{en cualquier otro caso.} \end{cases}$$

*Se denota* $f(x,\vec{y}) = \mu i \leq |x|[g(i,\vec{y}) = 0]$.

*Análogamente decimos que* $f$ *esta definida por maximización fuertemente acotada (*SBMAX*) si:*

$$f(x,\vec{y}) = \begin{cases} \text{máx}\{i \leq |x| : g(i,\vec{y}) = 0\}, & \textit{si existe,} \\ 0, & \textit{en cualquier otro caso.} \end{cases}$$

**Lema 4.1.3.** $A_0$ *es cerrada bajo* SBMIN *y* SBMAX.

**Demostración.** Sea una función $k$ dada mediante CRN de la forma:

$$\begin{cases} k(0,\vec{y}) = 0 \\ k(s_i(z),\vec{y}) = s_{h(z,\vec{y})}(k(z,\vec{y})) \end{cases}$$

donde

$$h(z,\vec{y}) = \begin{cases} 0, & \text{si } (\exists x \leq |z|)[g(x,\vec{y}) = 0] \\ 1, & \text{en cualquier otro caso.} \end{cases}$$

La función $k$ calculará un entero cuya representación binaria viene dada por un bloque de 1 seguido de un bloque de ceros. La longitud del primer bloque coincidirá con el valor mínimo tal que $g(x,\vec{y}) = 0$. Con esto en mente, podemos dar $f(x,\vec{y}) = \mu i \leq |x|[g(i,\vec{y}) = 0]$ definiéndola como:

$$f(x,\vec{y}) = \begin{cases} 0, & \text{si } (g(0,\vec{y}) = 0) \vee (\neg(\exists i \leq |x|)[g(i,\vec{y}) = 0]) \\ |rev(k(s_1(x),\vec{y}))| & \text{en cualquier otro caso.} \end{cases}$$

En el caso de la maximización, es decir, si:

$$f(x,\vec{y}) = \begin{cases} \text{máx}\{i \leq |x| : g(i,\vec{y}) = 0\}, & \text{si existe,} \\ 0, & \text{en cualquier otro caso.} \end{cases}$$

Se puede dar la función $k(z,\vec{y})$ tal que:

$$\begin{cases} k(0,\vec{y}) = 0 \\ k(s_i(z),\vec{y}) = s_{\overline{sg}(g(|z|,\vec{y}))}(k(z,\vec{y})) \end{cases}$$

y observar que $f(x,\vec{y}) = |k(s_1(x),\vec{y})|$. □

### 4.1.5 *Predicado orden, suma y resta.*

Como primer paso para conseguir en $A_0$ el predicado orden ($x \leq y$), la suma y la resta, se van a introducir las funciones MSP y LSP, las cuales serán muy utilizadas a lo largo del presente trabajo.

**Definición 4.1.4.** *Se define la función* parte más significativa *(MSP) usando* CRN *como:*

$$\begin{cases} \text{MSP}(0,y) = 0 \\ \text{MSP}(s_i(x),y) = s_{\text{BIT}(y,s_i(x))}(\text{MSP}(x,y)). \end{cases}$$

De esta forma, MSP devolverá un natural cuya representación binaria se corresponderá con el segmento inicial de $x$ en notación binaria truncando a partir del dígito anterior a la posición $y$. Por ejemplo, sean $y = 3$ y $x = 18$, si se tiene en cuenta que la notación binaria de 18 es $10010_2$ entonces:

$$\begin{aligned} \text{MSP}(18,3) &= \text{MSP}(10010,3) \\ &= s_{\text{BIT}(3,10010)(\text{MSP}(1001,3))} \\ &= s_0(s_{\text{BIT}(3,1001)}(\text{MSP}(100,3))) \\ &= s_0(s_1(s_{\text{BIT}(3,100)}(\text{MSP}(10,3)))) \\ &= s_0(s_1(s_0(s_{\text{BIT}(3,10)}(\text{MSP}(1,3))))) \\ &= s_0(s_1(s_0(s_0(s_{\text{BIT}(3,1)}(\text{MSP}(0,3)))))) \\ &= s_0(s_1(s_0(s_0(0)))) = 10_2 = 2. \end{aligned}$$

**Definición 4.1.5.** *Se define la función* parte menos significativa *(LSP) como:*

$$\text{LSP}(x, y) = \text{MSP}(rev(\text{MSP}(rev(s_1(x)), |\text{MSP}(x, y)|)), 1).$$

Como es de esperar, LSP calcula el número cuya representación binaria coincide con el segmento final de $x$ (en notación binaria) cuya longitud es $y$. Recurriendo al mismo ejemplo anterior:

$$\begin{aligned}
\text{LSP}(18, 3) &= \text{LSP}(10010, 3) \\
&= \text{MSP}(rev(\text{MSP}(rev(s_1(10010)), |\text{MSP}(10010, 3)|)), 1) \\
&= \text{MSP}(rev(\text{MSP}(rev(100101), |10|)), 1) \\
&= \text{MSP}(rev(\text{MSP}(101001, 2)), 1) \\
&= \text{MSP}(rev(1010), 1) \\
&= \text{MSP}(0101, 1) = 010_2 = 2.
\end{aligned}$$

Se detallan ahora tres predicados que nos ayudarán a continuar el estudio de las capacidades de $A_0$:

- $x\text{By}$: $x$ es el comienzo de $y$, si la notación binaria de $x$ coincide con el inicio de la representación binaria de $y$ (de izquierda a derecha). Se tiene que $x\text{By} \in (A_0)_*$:

$$x\text{By} \iff \begin{cases} (x = 0) \quad \vee \\ \big[(\forall i \leq |x|)[\text{BIT}(i, rev(s_1(x))) = \text{BIT}(i, rev(s_1(y)))] \wedge \\ \qquad (y > 0) \wedge (x > 0)\big]. \end{cases}$$

- $x\text{Ey}$: $x$ es final de $y$, siempre que la notación binaria de $y$ finalice con un segmento igual a la notación binaria de $x$. Puede expresarse en $(A_0)_*$ de la forma:

$$x\text{Ey} \iff \begin{cases} (x = 0) \quad \vee \\ \big[(\forall i \leq |x|)[\text{BIT}(i, x) = \text{BIT}(i, y)] \wedge (y > 0) \wedge (x > 0)\big] \end{cases}$$

  En este punto no podemos dejar pasar por alto un detalle, tanto en este como en el predicado anterior, hemos utilizado un predicado donde se usa la igualdad, $\text{BIT}(i, x) = \text{BIT}(i, y)$, aunque aún no hemos probado que esto sea posible en $A_0$. En este caso particular, podemos tomar esta licencia ya que la función BIT sólo puede tomar lo valores 0 ó 1, de forma que podemos entender:

$$\begin{aligned}
\text{BIT}(i, x) = \text{BIT}(i, y) \quad \equiv \quad & \big[(\text{BIT}(i, x) = 0 \wedge \text{BIT}(i, y) = 0) \vee \\
& (\text{BIT}(i, x) \neq 0 \wedge \text{BIT}(i, y) \neq 0)\big].
\end{aligned}$$

- $x\text{Py}$: $x$ forma parte de $y$, si la representación binaria de $x$ coincide con algún segmento que forme parte de la notación binaria de $y$. Formalmente, podemos definir este predicado en $(A_0)_*$ como:

$$x\text{Py} \iff \exists i < |y|(x\text{E}\text{MSP}(y, i)).$$

**Lema 4.1.6.** *$(A_0)_*$ es cerrado bajo los cuantificadores del tipo es-parte-de, es decir, $(\exists x B y)$, $(\exists x E y)$, $(\exists x P y)$, $(\forall x B y)$, $(\forall x E y)$ y $(\forall x P y)$*

**Demostración.** Sea $R(x, \vec{z})$ un predicado de $A_0$, entonces:

$$\begin{aligned}
(\exists x B y) R(x, \vec{z}) &\iff \exists j < |y|(R(\mathrm{MSP}(y, j), \vec{z})),\\
(\exists x E y) R(x, \vec{z}) &\iff \exists j < |y|(R(\mathrm{LSP}(y, j), \vec{z})),\\
(\exists x P y) R(x, \vec{z}) &\iff \exists u B y\ \exists x E u(R(x, \vec{z})).
\end{aligned}$$

Para los cuantificadores existenciales $(\forall x B y)$, $(\forall x E y)$ y $(\forall x P y)$, bastará utilizar convenientemente la función $\overline{sg}(x)$ en los razonamientos anteriores.

□

**Proposición 4.1.7.** *El predicado orden $(x \leq y)$ y las funciones la suma $(+)$ y resta $(\dot{-})$ pertenecen a $A_0$.*

**Demostración.** Se prueba como consecuencia inmediata del lema 4.1.6:

La función característica del predicado orden $P(x, y) = (x \leq y)$ realiza el siguiente cálculo:

$$c_{\leq}(x, y) = \begin{cases} 1, & \text{si } |x| < |y|,\\ 0, & \text{si } |x| > |y|,\\ c_{P(x,y)}, & \text{si } |x| = |y|. \end{cases}$$

En $A_0$, se define $P(x, y) = (x \leq y)$ como

$$P(x, y) = (\exists u B x)[u B y \wedge \mathrm{BIT}(|x| \dot{-} |u| \dot{-} 1, y) = 1 \wedge \mathrm{BIT}(|x| \dot{-} |u| \dot{-} 1, x) = 0].$$

Quedará probado que $P(x, y) \in A_0$ simplemente observando algunos detalles que aparecen por primera vez en esta función:

- El predicado $|x| < |y|$ tiene como función característica $sg(\mathrm{MSP}(y, |x|))$.
- El predicado $|x| = |y|$ tiene como función característica:

$$c_{|x|=|y|}(x, y) = \begin{cases} \overline{sg}(\mathrm{MSP}(y, |x|)), & \text{si } \neg(|y| < |x|),\\ 0, & \text{si } |y| < |x|. \end{cases}$$

- La función $|x| \dot{-} |u| \dot{-} 1$ se puede expresar en $A_0$ como $|\mathrm{MSP}(\mathrm{MSP}(x, |u|), 1)|$.
- Los predicados $\mathrm{BIT}(|x| \dot{-} |u| \dot{-} 1, y) = 1$ y $\mathrm{BIT}(|x| \dot{-} |u| \dot{-} 1, x) = 0$ tiene funciones características $\mathrm{BIT}(|x| \dot{-} |u| \dot{-} 1, y)$ y $\overline{sg}(\mathrm{BIT}(|x| \dot{-} |u| \dot{-} 1, y))$, respectivamente.

Podemos dar la suma, $x + y$, en $A_0$ definiéndola por partes:

$$x + y = \begin{cases} sum(x, y, s_1(s_1(x))) & \text{si } y \leq x,\\ sum(x, y, s_1(s_1(y))) & \text{en caso contrario,} \end{cases}$$

donde la función $sum(x,y,z)$ devuelve el entero cuya representación binaria es la suma de $x$ e $y$ truncada por la derecha en el bit $|z|$. Se puede definir usando recursión por concatenación sobre la notación:

$$\begin{cases} sum(x,y,0) = 0, \\ sum(x,y,s_i(z)) = s_{\text{BIT}(|z|,x)\oplus\text{BIT}(|z|,y)\oplus acarreo(x,y,z)}(sum(x,y,z)) \end{cases}$$

De nuevo, es necesario realizar unas apreciaciones sobre algunas funciones y predicados usados, con especial atención a $acarreo(x,y,z)$, que calculará el acarreo de una suma clásica en el $|z|$–ésimo bit:

- La función $\oplus$ se refiere a función lógica xor, que puede darse en $A_0$ como:

$$x \oplus y = cond(x, cond(x,0,1), cond(x,1,0)).$$

- Calcular el acarreo del $|z|$–ésimo bit de $x+y$, entendiendo ese acarreo en la suma de números en representación binaria, conlleva conocer si el acarreo se genera en ese bit concreto (GEN) o se propaga un acarreo que venía de un bit anterior (PROP). Los siguientes predicados se encargan de señalar ambos casos:

$$\text{GEN}(x,y,z) = [(\text{BIT}(|z|,x) = 1 \wedge \text{BIT}(|z|,y) = 1)]$$
$$\text{PROP}(x,y,z) = [(\text{BIT}(|z|,x) = 1 \vee \text{BIT}(|z|,y) = 1)]$$

  Se puede ahora describir la función $acarreo$ en $A_0$:

$$acarreo(x,y,z) = \begin{cases} 1 & \text{si } (\exists u \text{B} z)[\text{GEN}(x,y,u) \wedge \\ & \qquad (\forall v \text{B} z)[|v| > |u| \to \text{PROP}(x,y,v)]]), \\ 0 & \text{en caso contrario.} \end{cases}$$

  El predicado que condiciona el primer caso será válido si en un bit $|u|$ menor que $|z|$ se genera un acarreo en $x+y$ y en los bits entre $|z|$ y $|u|$ sólo ocurren propagaciones.

- El último detalle surge del uso de la implicación lógica en la definición de la función $acarreo$. Si la implicación $P_1(\vec{z}) \to P_2(\vec{z})$ la vemos como una conjunción, $\neg P_1(\vec{z}) \vee P_2(\vec{z})$, es fácil ver que está en $(A_0)_*$:

$$c_{P_1 \vee P_2}(\vec{z}) = \begin{cases} 1 & \text{si } P_2(\vec{z}), \\ c_{\neg P_2}(\vec{z}) & \text{en caso contrario.} \end{cases}$$

La última función es la función resta $x \dot{-} y$, entendiéndola como $\text{máx}(x-y,0)$. Para probar que dicha función pertenece a $A_0$ basta seguir punto por punto la estrategia dada para la suma, donde únicamente habrá que cambiar las definiciones de las funciones que generan o propagan acarreos, GEN o PROP, respectivamente. Resumiendo quedaría:

$$x \dot{-} y = \begin{cases} resta(x,y,x) & \text{si } y \leq x, \\ 0 & \text{en caso contrario.} \end{cases}$$

donde:

$$\begin{cases} \text{resta}(x,y,0) = 0, \\ \text{resta}(x,y,s_i(z)) = s_{\text{BIT}(|z|,x)\oplus\text{BIT}(|z|,y)\oplus\text{acarreo}(x,y,z)}(\text{resta}(x,y,z)). \end{cases}$$

Queda dar las definiciones de acarreo, GEN y PROP:

$$\text{acarreo}(x,y,z) = \begin{cases} 1 & \text{si } (\exists u \mathrm{B} z)[\text{GEN}(x,y,u) \wedge \\ & \qquad (\forall v \mathrm{B} z)[|v| > |u| \rightarrow \text{PROP}(x,y,v)]]), \\ 0 & \text{en caso contrario.} \end{cases}$$

$$\text{GEN}(x,y,z) = [(\text{BIT}(|z|,x) = 0 \wedge \text{BIT}(|z|,y) = 1)],$$

$$\text{PROP}(x,y,z) = \big[[(\text{BIT}(|z|,x) = 0 \wedge \text{BIT}(|z|,y) = 0] \vee \text{BIT}(|z|,y) = 1)\big].$$

□

## 4.2 CODIFICACIÓN DE UNA COMPUTACIÓN DE UNA RATM.

Se persigue en este apartado lograr una aritmetización del comportamiento de una máquina de Turing de acceso aleatorio (recordamos: RATM). Por tanto, si llamamos M a una RATM, será necesario alcanzar:

- La codificación de cada una de las configuraciones de M.
- Una función, $\text{INICIAL}_M(x)$, que devuelva la codificación de una configuración inicial de M a partir de una entrada x.
- Un predicado, $\text{NEXT}_M(x,\alpha,\beta)$, que se cumpla sobre las configuraciones $\alpha$ y $\beta$ de M sobre una entrada x si y sólo si $\beta$ es una configuración siguiente a $\alpha$.

Codificar una configuración de una RATM no es más que codificar una tupla de número naturales (definición de configuración 2.1.5) de forma que parece lógico codificar un par, para después extender este procedimiento a tuplas de cualquier dimensión. Por tanto, se define la función $\tau$ como sigue:

$$\tau(x,y) = (2^{\text{máx}(|x|,|y|)} + x) * (2^{\text{máx}(|x|,|y|)} + y).$$

Ver que $\tau(x,y)$ está en $A_0$ no es más que observar que:

$$2^{\text{máx}(|x|,|y|)} = \text{cond}(\text{MSP}(x,|y|), \text{pad}(1,y), \text{pad}(1,x)).$$

Suponiendo que las representaciones binarias de dos números naturales x e y son de la forma $a_n a_{n-1} \dots a_1 a_0$ y $b_n b_{m-1} \dots b_1 b_0$, con $|x| > |y|$, la codificación de ambas palabras sería un $z \in \mathbb{N}$ tal que $\tau(x,y) = z$ y cuya notación binaria es:

$$\overbrace{1\underbrace{a_n a_{n-1} \dots a_1 a_0}_{x}}^{|x|+1} \overbrace{10\dots0\underbrace{b_n b_{m-1} \dots b_1 b_0}_{y}}^{|x|+1}.$$

Por ejemplo, $\tau(3,4) = 188$ y las representaciones binarias de 3, 4 y 188 son 11, 100 y 10111100 respectivamente.

Este procedimiento permite recuperar de forma inequívoca los dos números codificados por la función $\tau(x,y)$ usando las proyecciones izquierda $\pi_1$ y derecha $\pi_2$ definidas como sigue:

$$\begin{aligned}\pi_1(z) &= \text{TL}(\text{MSP}(z, \lfloor \frac{|z|}{2} \rfloor)) \\ \pi_2(z) &= \text{TL}(\text{LSP}(z, \lfloor \frac{|z|}{2} \rfloor))\end{aligned}$$

donde TL y TR son funciones que truncan el bit más a la izquierda y el más a la derecha (respec.) de la representación binaria de su entrada, es decir:

$$\begin{aligned}\text{TR}(x) &= \text{MSP}(x,1) \\ \text{TL}(x) &= \text{LSP}(x,|\text{TR}(x)|) = \text{TR}(rev(\text{TR}(rev(s_1(x)))))\end{aligned}$$

Aplicando $\tau$ las veces necesarias se consigue codificar una tupla $(x_1,\ldots,x_n)$, de modo que la codificación $\tau_n(x_1,\ldots,x_n)$ vendrá dada por:

$$\tau_n(x_1,\ldots,x_n) = \tau(x_1,\tau(x_2,\tau(\ldots\tau(x_{n-1},x_n)\ldots))).$$

Tal y como se define en 2.1.5 una configuración de una máquina de Turing será un elemento de $Q \times (\Sigma \cup \{B\})^* \times (\Gamma \cup \{B\})^{*k} \times \mathbb{N}^{k+1}$, que representa el estado actual de la máquina, el contenido de la cinta de entrada y las $k$ cintas de trabajo y la posición de las $k+1$ cabezas. Si dicha máquina es de acceso aleatorio, no serán necesarias ni la cinta de entrada ni la cabeza lectora de la misma. Por contra, la máquina contará con dos cintas más, la que contiene el índice del bit de la entrada solicitado y la que contiene la respuesta a dicha consulta. Por tanto, las configuraciones de una RATM son elementos de $Q \times (\Gamma \cup \{B\})^{*k+2} \times \mathbb{N}^{k+2}$.
Dada una configuración $(q,u_1,\ldots,u_{k+2},n_1,\ldots,n_{k+2})$, una modificación útil en las mismas antes de codificarlas es verlas como $(q,l_1,r_1,\ldots,l_{k+2},r_{k+2})$, donde cada $l_i$ representa el contenido de la porción izquierda de la cinta $i$–ésima hasta la posición de la cabeza de lectura (incluida ésta) y $r_i$ el reverso de la porción derecha a partir de dicha posición.

Si los elementos el alfabeto $\Gamma \cup \{B\}$ se codifican usando el alfabeto $\{0,1\}$, una configuración de una RATM puede codificarse por:

$$\tau_{2k+5}(q,l_1,r_1,\ldots,l_{k+2},r_{k+2}).$$

Quedando las configuraciones totalmente representadas en $A_0$, resta ver qué ocurre con la función $\text{INICIAL}_M$ y el predicado $\text{NEXT}_M$ descritos unas líneas más arriba.

**Lema 4.2.1.** *La función* $\text{INICIAL}_M$ *y el predicado* $\text{NEXT}_M$ *pertenecen al álgebra:*

$$[0, I, s_0, s_1, \text{BIT}, |x| : \text{COMP}, \text{CRN}] \subseteq A_0.$$

**Demostración.** Se notará a lo largo de la demostración al álgebra de funciones $[0, I, s_0, s_1, \text{BIT}, |x| : \text{COMP}, \text{CRN}]$ como $\mathcal{F}$. Al ser el conjunto Q de estados de la máquina finito, si se supone que el número de estados es $m$, existe una biyección entre el conjunto $\{0, \ldots, m\}$ y Q, de forma que cada estado $q_i$ queda codificado por un $k \in \mathbb{N}$ con $k \leq m$. Entonces:

$$\text{INICIAL}_M(x) = \tau_{2k+5}(q_0, b, \ldots, b)$$

donde $q_0$ es el número natural que representa al estado inicial y $b$ es el natural cuya notación binaria codifica el símbolo B.

Para el predicado $\text{NEXT}_M(x, \alpha, \beta)$ se necesitarán algunas funciones y predicados auxiliares no excesivamente complicadas. Suponiendo que tanto $\alpha$ como $\beta$ son codificaciones de configuraciones de M de la forma dada en 2.1.5, en primer lugar se obtienen $2k+5$ funciones que decodifican cada una de las componentes de una configuración, es decir, si:

$$\begin{aligned} \alpha &= \tau_{2k+5}(q, l_1, r_1, \ldots, l_{k+2}, r_{k+2}) \\ \beta &= \tau_{2k+5}(q, l'_1, r'_1, \ldots, l'_{k+2}, r'_{k+2}) \end{aligned}$$

se tendrán $2k+5$ funciones de la forma:

$$\begin{aligned} \text{COOR}_1(\alpha) &= \pi_1(\alpha) = q \\ \text{COOR}_2(\alpha) &= \pi_2(\pi_1(\alpha)) = l_1 \\ &\vdots \quad\quad \vdots \\ \text{COOR}_{2k+5}(\alpha) &= \pi_2(\ldots \pi_2(\pi_1(\alpha)) \ldots) = r_{k+2}. \end{aligned}$$

Tomando ahora $\delta \in \Delta$, tal y como se describe en 2.1.7, se tiene que:

$$\delta = (q_\delta, \sigma_1, \ldots, \sigma_{k+2}, r_\delta, \tau_1, \ldots, \tau_{k+2}, d_0, d_1, \ldots, d_{k+2}) \in \Delta.$$

El predicado $P_M(\alpha, \beta, \delta)$, que indicará que $\alpha \vdash_M \beta$ siguiendo la instrucción $\delta$, se puede describir en $[0, I, s_0, s_1, \text{BIT}, |x| : \text{COMP}, \text{CRN}]$ de una manera un tanto tediosa, comprobando cada elemento de $\delta$:

$$\begin{aligned} P_M(\alpha, \beta, \delta) \equiv\ & [\text{COOR}_1(\alpha) = \text{COOR}_1(\delta)] \wedge \\ & [\text{LSP}(\text{COOR}_2(\alpha), 1) = \text{COOR}_2(\delta)] \wedge \\ & [\text{LSP}(\text{COOR}_4(\alpha), 1) = \text{COOR}_3(\delta)] \wedge \ldots \wedge \\ & [\text{LSP}(\text{COOR}_{2k+4}(\alpha), 1) = \text{COOR}_{k+3}(\delta)] \wedge \\ & [\text{COOR}_1(\beta) = \text{COOR}_{k+4}(\delta)] \wedge \\ & [\text{LSP}(\text{COOR}_2(\beta), 1) = \text{COOR}_{k+5}(\delta)] \wedge \\ & [\text{LSP}(\text{COOR}_4(\beta), 1) = \text{COOR}_{k+5}(\delta)] \wedge \ldots \wedge \\ & [\text{LSP}(\text{COOR}_{2k+4}(\beta), 1) = \text{COOR}_{2k+6}(\delta)] \wedge \\ & [\text{MOV}(\text{COOR}_2(\beta), \text{COOR}_2(\alpha), \text{COOR}_{2k+7}(\delta))] \wedge \\ & [\text{MOV}(\text{COOR}_4(\beta), \text{COOR}_4(\alpha), \text{COOR}_{2k+8}(\delta))] \wedge \ldots \wedge \\ & [\text{MOV}(\text{COOR}_{2k+4}(\beta), \text{COOR}_{2k+4}(\alpha), \text{COOR}_{3k+8}(\delta))], \end{aligned}$$

donde vamos a definir el predicado MOV de forma que se cumplirá si los movimientos de las cabezas que indican las configuraciones $\alpha$ y $\delta$ se corresponde que señala $\delta$:

$$\text{MOV}(x,y,z) \iff \begin{cases} |x| > |y|, & \text{si } z = 1 \\ |x| = |y|, & \text{si } z = 0 \\ |x| < |y|, & \text{en otro caso.} \end{cases}$$

Para finalizar, al ser $\Delta$ un conjunto finito, si se supone que $\Delta = \{\delta_1, \ldots, \delta_n\}$ se puede obtener el predicado $\text{NEXT}_M(x,\alpha,\beta)$:

$$\text{NEXT}_M(x,\alpha,\beta) \equiv P_M(\alpha,\beta,\delta_1) \vee \ldots \vee P_M(\alpha,\beta,\delta_n).$$

□

El siguiente paso en la aritmetización de una máquina de Turing debe ser la codificación, mediante elementos de $A_0$, de secuencias de números que representen a las configuraciones tal y como se acaba de ver. Para ello, en primer lugar se va a dar una noción intuitiva del razonamiento a seguir en el teorema 4.2.2 para lograr dicha codificación.

Se parte de generalizar la idea de la función $\tau$ a una secuencia de números. Por ejemplo, para codificar la secuencia $(4,8,3,0)$ primero se calcula el tamaño del bloque $\text{BS} = \text{máx}\{|4|,|8|,|3|,|0|\} + 1$ para después calcular $t \in \mathbb{N}$ cuya representación binaria es:

$$1\underbrace{0100}_{4}1\underbrace{1000}_{8}1\underbrace{0011}_{3}1\underbrace{0000}_{0}.$$

Si $\ell$ es la longitud de la secuencia, entonces $|t| = \ell \cdot \text{BS}$. Con $t$ y $\ell$, se aplica la función de nuevo $\tau$ (para aridad 2) obteniendo la codificación:

$$z = \langle 4,8,3,0\rangle = \tau(t,\ell).$$

En el caso de la decodificación, si $z$ es el resultado de codificar una secuencia, la función correspondiente $\beta(i,z)$ de Gödel, que nos devuelve la palabra codificada en $z$ en la posición posición $i$, vendrá dada por:

$$\begin{cases} \beta(0,z) = \pi_2(z) = \ell = 4 \\ \beta(i,z) = \text{LSP}(\text{MSP}(\pi_1(z),(\ell - i)\cdot \text{BS}), \text{BS} - 1) \end{cases}$$

donde se calcula $\text{BS} = \lfloor |\pi_1(z)|/\pi_2(z)\rfloor = \lfloor 20/4\rfloor = 5$ y $i \in \{1,\ldots,4\}$. Este razonamiento choca con la probable no pertenencia de la división y la multiplicación a $A_0$. Como se verá en la demostración del siguiente teorema, este obstáculo se salva al ser posibles las divisiones y productos por potencias de 2 en $A_0$, con lo que bastará con tomar un tamaño de bloque BS que sea potencia de 2 y que aporte el suficiente espacio para codificar cada entero de la secuencia.

**Teorema 4.2.2.** *Sea $f \in A_0$, entonces existe $g \in A_0$ tal que para cada $x$ se cumple*

$$g(x,\vec{y}) = \langle f(0,\vec{y}),\ldots,f(|x|-1,\vec{y})\rangle.$$

**Demostración.** Esta demostración es totalmente constructiva, se pretende dar una función $g$ que permita codificar la secuencia $(f(0,\vec{y}),\dots,f(|x|-1,\vec{y}))$ por un $z = \langle f(0,\vec{y}),\dots,f(|x|-1,\vec{y})\rangle$ mediante un procedimiento similar al dado en la introducción a este teorema.

Como ya se ha comentado, la definición del tamaño de bloque BS de la codificación será fundamental para conseguir que está se mantenga en $A_0$, llegando a que este objetivo será alcanzable si BS es potencia de 2. Si se tiene en cuenta que

$$|x| = \lceil \log(x+1) \rceil = 2^{\log(\lceil \log(x+1) \rceil)} \leq 2^{\lceil \log(\lceil \log(x+1) \rceil + 1) \rceil} = 2^{\|x\|}$$

será razonable que se defina el tamaño de bloque como:

$$BS = \max\{2^{\|f(i,\vec{y})\|} : 0 \leq i \leq |x|-1\}.$$

Hay que hacer un par de indicaciones antes de continuar: la primera es que la función $2^{\|x\|}$ puede darse en $A_0$ como $|x|\#1$, y la segunda es que ocurre lo mismo con la maximización utilizada, es decir, si $g(x,\vec{y}) \in A_0$ entonces:

$$\max\{g(i,\vec{y}) : i \leq |x|\} = g((\mu i \leq |x|)(\forall j \leq |x|)[g(j,\vec{y}) \dot{-} g(i,\vec{y}) = 0)],\vec{y}).$$

Se define ahora la función $q(z,\vec{y})$, que para $0 \leq z \leq BS \cdot |x|$ generará cada bit de la representación binaria de la codificación de la secuencia $(f(0,\vec{y}),\dots,f(|x|-1,\vec{y}))$:

$$q(z,\vec{y}) = \begin{cases} 1, & \text{si } |x| \text{ (mód BS)} = 0 \\ BIT((BS \dot{-} 1) \dot{-} (|z| \text{ (mód BS)}), f(\frac{|z|}{BS} + 2, \vec{y})), & \text{en cualquier otro caso.} \end{cases}$$

La siguiente función, $h(z,\vec{y})$, calcula el número cuya representación binaria está compuesta por cada uno de los bits que calcula la función $q$ usando CRN:

$$\begin{cases} h(0,\vec{y}) = 0 \\ h(s_i(z),\vec{y}) = s_{q(z,\vec{y})}(h(z,\vec{y})). \end{cases}$$

Por lo tanto, la codificación quedará completada aplicando la función $pairing$ para añadir a la misma la longitud de la secuencia:

$$g(x,\vec{y}) = \tau(h(BS \cdot |x|,\vec{y}),|x|).$$

Se tiene, por tanto, que $g(x,\vec{y}) = \langle f(0,\vec{y}),\dots,f(|x|-1,\vec{y})\rangle$, pero en la construcción de la misma se ha multiplicado, dividido y hecho módulo con BS por lo que, para que $g \in A_0$, estas tres funciones deben estar en dicha álgebra. Dicho de otra forma, dado que BS es una potencia de 2, bastará con dar:

- La función $f(x,y) = x \cdot 2^y$ se define en $A_0$ como:

$$f(x,y) = \begin{cases} h(0,x) = x \\ h(s_i(y),x) = s_0(h(y,x)). \end{cases}$$

- $f(x,y) = \lfloor \frac{x}{2^y} \rfloor$ es exactamente la función $\text{MSP}(x,y)$.
- Igualmente, $f(x,y) = x \pmod{2^y}$ puede darse en $A_0$ mediante $\text{LSP}(x,y)$.

Con esto finaliza la construcción de la función de codificación $g(x,\vec{y})$ en $A_0$. □

Toda está construcción debe ser complementada con una función $\beta(i,z) \in A_0$ que permita recuperar los elementos de una secuencia codificada $z = \langle a_1,\ldots,a_n\rangle$, para ello se llama $\ell h(z)$ al número de elementos de la secuencia, el cual es fácil de obtener:

$$\ell h(z) = \beta(0,z) = \begin{cases} \pi_2(z), & \text{si } z \text{ codifica un par,} \\ 0, & \text{en cualquier otro caso.} \end{cases}$$

Con esto, obtener el elemento $i$-ésimo de la secuencia codificada por $z$, para $1 \leq i \leq \beta(0,z)$, no será más que aplicar:

$$\beta(i,z) = \text{LSP}(\text{MSP}(\pi_1(z),(\ell h(z) \dot{-} i)\cdot\left\lfloor \frac{|\pi_1(z)|}{\ell h(z)} \right\rfloor), \left\lfloor \frac{|\pi_1(z)|}{\ell h(z)} \right\rfloor \dot{-} 1).$$

Queda como pieza extraña de esta ecuación la expresión $\lfloor \frac{|\pi_1(z)|}{\ell h(z)} \rfloor$, que no es más que el tamaño de bloque $\text{BS}$ que se usa en la demostración del teorema anterior. Pero, dado que este es una potencia de 2, que ya se tiene el número de elementos de la secuencia (i.e. $\ell h(z)$) y que $|\pi_1(z)| = \text{BS}\cdot\ell h(z)$ se puede calcular el tamaño de bloque en $A_0$ mediante $\text{BS} = 2^m$ donde:

$$m = (\mu x \leq ||\pi_1(z)||)[\text{MSP}(|\pi_1(z)|,x) = \ell h(z)].$$

## 4.3 EQUIVALENCIA ENTRE LA CLASE $\mathcal{F}\text{LH}$ Y $A_0$.

Se introducen seguidamente dos lemas que culminarán los preparativos para probar que $A_0 = \mathcal{F}\text{LH}$:

**Lema 4.3.1.** *Para todo* $k, m \in \mathbb{N}$, *con* $k, m > 1$ *se tiene:*

$$\text{DTIMESPACE}(\log^k(n), \log^{1-1/m}(n)) \subseteq A_0.$$

**Demostración.** Sea $M$ una RATM con tiempo $\log^k(n)$ y espacio $\log^{1-1/m}(n)$. Para cada $i \leq m\cdot k$ se define el predicado $\text{NEXT}_{M,i}$ de la forma:

$$\text{NEXT}_{M,i}(x,c,d) \quad \Leftrightarrow \quad [\ d \text{ es una configuración alcanzable desde } c \text{ en a lo sumo } \log^{i/m}(n) \text{ pasos}].$$

siendo $c$ y $d$ codificaciones de configuraciones de una computación de $M$ sobre la entrada $x$ y $n = |x|$. Con la definición de este predicado se tiene que $M$ aceptará una palabra $x$ si y sólo si $\text{NEXT}_{M,m\cdot k}(x,c,d)$ se cumple, tomando $c$ y $d$ como las codificaciones de una configuración inicial y una configuración final de aceptación, y siempre y cuando en dicho predicado se apliquen las cotas necesarias para que el espacio utilizado sea sublogarítmico. Con todo esto se quiere

reseñar, en definitiva, que mostrando que el predicado $\text{NEXT}_{M,m\cdot k}(x, c, d)$ está en $A_0$ el lema quedará probado a su vez.

Se demuestra por inducción sobre $i$ que $\text{NEXT}_{M,i}$ está en $A_0$. El caso base de esta prueba, es decir, para $i = 0$, se refiere a:

$$\text{NEXT}_{M,0}(x, c, d) \Leftrightarrow [\ d \text{ es la configuración siguiente a } c \text{ en a lo sumo un paso}]$$

que es exactamente el resultado obtenido en el lema 4.2.1 y, por tanto, $\text{NEXT}_{M,0} \in A_0$.

Si se supone ahora, como paso de inducción, que $\text{NEXT}_{M,i} \in A_0$, el predicado $\text{NEXT}_{M,i+1}(x, c, d)$ puede ser definido en $A_0$ como:

$$\begin{aligned}(\exists s \leq |x|^3)(\forall j < ||x||^{1/m} - 1) \quad & [s = \langle s_0, \ldots, s_{||x||^{1/m}-1}\rangle \wedge \\ & c = s_0 \wedge d = s_{||x||^{1/m}-1} \wedge \\ & \text{NEXT}_{M,i}(x, s_j, s_{j+1})].\end{aligned}$$

Con esto quedaría probado el enunciado del lema, aunque sería conveniente aclarar algunos detalles de este predicado:

- La cota utilizada en el cuantificador universal viene dada directamente por el tiempo $\log^k(n)$ de M. En cada $i$ la codificación de la secuencia $s$ tendrá estrictamente menos de $||x||^{1/m}$ elementos, es decir, $(\lceil \log(n+1)\rceil)^{1/m}$. De esta forma se asegura que la secuencia $s$ tenga a lo sumo una longitud de $\log^{1/m}(n)$ y así, cuando $i = m \cdot k$, el predicado indicará que se dan como máximo $(\log^{1/m}(n))^{m\cdot k} = \log^k(n)$ pasos.

- Para la cota del cuantificador existencial se tiene en cuenta, además, que el espacio que puede usar cada $s_j$ debe ser menor o igual a $\log^{1-1/m}(n)$. Con esto se llega a que:

  $$|s_j| \leq \log^{1-1/m}(|x|) \leq (\lceil \log(|x|+1)\rceil)^{1-1/m} \leq ||x||^{1-1/m}.$$

  Por lo tanto, la longitud de $s$, el entero que codifica la secuencia:

  $$\langle s_0, \ldots, s_{||x||^{1/m}-1}\rangle,$$

  será como máximo el producto de elementos de codifica por el espacio que pueden ocupar estos, es decir:

  $$|s| \leq ||x||^{1/m} \cdot (||x||^{1-1/m} + 1) \leq 2 \cdot ||x||.$$

  Se llega entonces a que, de existir $s$, con seguridad será menor o igual que $|x|^3$, que a su vez se puede calcular como $||x\#x|\#x|$.

- El predicado está compuesto por una conjunción de cuatro elementos que indican respectivamente: $s$ codifica una secuencia de a lo sumo $||x||^{1/m}$ elementos, el primer elemento de la secuencia coincide con $c$, el último elementos lo hace con $d$ y, para todo $j < ||x||^{1/m} - 1$, $s_{j+1}$ es la configuración siguiente a $s_j$ en a lo sumo $i$ pasos. □

**Lema 4.3.2.** *La clase* $\text{DSPACE}(\log\log(n))$ *está contenido en* LH.

**Demostración.** Sea $M$ una RATM con tiempo polilogarítmico $\log^{\mathcal{O}(1)}(n)$ y espacio sublogarítmico $\log^{1-\varepsilon}(n)$. Usando la codificación dada en proposición 3.2.6, se puede definir una ATM para el predicado $\text{NEXT}_{M,0}(x, c, d)$, que señala cuando dos configuraciones son consecutivas (simplemente comparando bit a bit cada configuración). Además, dicha codificación puede ser generalizada para codificar una secuencia de la forma:

$$\langle x_0, \ldots, x_n \rangle = \sigma(|x_0|)11\ldots11\sigma(|x_n|)11x_1\ldots x_n.$$

Al igual que pasaba en 3.2.6, calcular el $i$-ésimo bit de la $j$-ésima palabra requiere un cálculo del orden de:

$$2 \cdot ||x_0|| + 2 + \ldots + 2 + 2 \cdot ||x_n|| + |x_1| + \ldots + |x_{j-1}| + i$$

con lo que es computable en tiempo logarítmico.

Teniendo esto en cuenta, se puede tomar una estrategia similar a la del lema anterior para probar que el predicado $\text{NEXT}_{M,i}(x, c, d)$, donde $c$ y $d$ son codificaciones de configuraciones, pertenece a LH.

Este desarrollo lleva a probar que $\text{DTIMESPACE}(\log^k(n), \log^{1-1/m}(n)) \subseteq \text{LH}$. Será suficiente ahora tener en cuenta que una computación de una ATM en $\text{DSPACE}(\log\log(n))$ puede tener, a lo sumo, $2^{k\cdot\log\log(n)} = \log^k(n)$ configuraciones para alguna constante $k$. De este hecho se deriva que:

$$\text{DSPACE}(\log\log(n)) \subseteq \text{DTIMESPACE}(\log^k(n), \log^{1-1/m}(n)) \subseteq \text{LH}$$

y, por tanto, $\text{DSPACE}(\log\log(n)) \subseteq \text{LH}$, tal y como se pretendía probar. □

Con todo el análisis previo hecho hasta ahora, en el que se han presentado una breve descripción de $A_0$ a través de sus componentes y propiedades, y una codificación en $A_0$ de las computaciones de una RATM, se reúnen las herramientas necesarias para demostrar que la clase $\mathcal{F}\text{LH}$ se corresponde con el álgebra de funciones $A_0$.

**Teorema 4.3.3.** $A_0 = \mathcal{F}\text{LH}$.

**Demostración.** Haremos la demostración por doble inclusión. Para probar la dirección $A_0 \subseteq \mathcal{F}\text{LH}$ se verá que tanto las funciones como los operadores básicos de $A_0$ pueden ser simulados por máquinas de LH:

- Haciendo uso, de nuevo, de la ya señalada proposición 3.2.6, es trivial comprobar que las funciones $0$, $s_0$, $s_1$ y $|x|$ son calculables por una RATM trabajando en tiempo logarítmico.
- Para $\text{BIT}(i, x)$, sea $M_1$ una RATM que recibe como entrada la palabra $BiBxB$. Por lo anterior, $M_1$ puede calcular $|i|$ para después copiar en la cinta de consulta el resultado de $i + |i| + 1$ de forma que en la cinta de respuesta quede guardado el $i$-ésimo bit de $x$. La única reflexión que necesita este apartado es notar que tan sólo se realizan sumas de enteros menores que el logaritmo de la longitud de la entrada.

- Para calcular el $i$-ésimo bit del $k$-ésimo argumento de la función proyección, $I_k^n(x_1,\ldots,x_n)$, una RATM $M_2$, que recibe como entrada la palabra $BiBx_1Bx_2B\ldots Bx_nB$, puede utilizar los estados universales y existenciales para encontrar los índices de los separadores B y, con esto, calcular la suma $i+k+\sum_{j<k}|x_j|$ que actuará de índice en la cinta de consulta.

- En el caso de la función #, la máquina $M_3$ calculará el $i$-ésimo bit de $x\#y = 2^{|x|\cdot|y|}$ dando como salida 1 si $i = |x|\cdot|y|$ y 0 en cualquier otro caso. Hay que apreciar que el producto $|x|\cdot|y|$ puede ser computado por una RATM en DSPACE$(\log\log(n))$ por lo que, aplicando el lema 4.3.2, se tiene que $\# \in \mathcal{F}$LH. Para dar una máquina que calcule dicho producto en espacio $\log\log(n)$ basta tomar como entrada BxByB y una máquina con cuatro cintas. Conforme se vayan calculando cada uno de los bits del producto, los índices de estos se puede almacenar en cada una de las cintas secuencialmente, separando estos índices por una B. No es difícil observar que esta máquina cumple con los requisitos de espacio.

- Para ver que $\mathcal{F}$LH es cerrado bajo composición supongamos que $f(x) = g(h_1(x),h_2(x))$ donde los bitgrafos de $g$, $h_1$ y $h_2$ son calculados por las ATM $M_g$, $M_{h_1}$ y $M_{h_2}$ respectivamente, todas ellas corriendo en tiempo logarítmico con un número constante de alternancias. En estas condiciones, es posible encontrar una máquina $M_f$ basada en $M_g$ que es capaz de computar la función $f$ en tiempo logarítmico.

  Teniendo en cuenta que $M_g$ tendrá como entradas palabras de la forma $By_1By_2B$, se puede definir $M_f$ como una ampliación de $M_g$ en la que la única diferencia será que cuando ésta solicite el $i$-ésimo bit de la entrada, $M_f$ calculará $|h_1(x)|$ y $|h_2(x)|$ y seguidamente ejecutará el siguiente algoritmo:

```
if i = 0 then
   return B
else if i ≤ ||h1(x)|| then
    return Mh1(i − 1, x)
else if i = ||h1(x)|| + 1 then
    return B
else if i ≤ ||h1(x)|| + 1 + ||h2(x)|| then
    return Mh2(i − ||h1(x)|| − 2, x)
else
    return B
```

  En estas instrucciones se utilizan inecuaciones para realizar comparaciones, lo cual se puede calcular en tiempo logarítmico pues:

$$\left[i \le |h_1(x)|\right] \Leftrightarrow \begin{cases} |i| < ||h_1(x)|| & \text{ó} \\ |i| = ||h_1(x)|| \text{ e } i \text{ precede lexicográficamente a } ||h_1(x)||. \end{cases}$$

  En cualquiera de los dos casos, como la comparación se realiza sobre $|h_1(x)|$ estamos trabajando en tiempo logarítmico.

Otro detalle a tener en cuenta en el algoritmo es el uso de máquinas de la forma $M_{h_1}(i-1,x)$, que no son más que simular el cálculo de $M_{h_1}(x)$ para seguidamente aplicar al resultado $M_{\text{BIT}}(i-1,x)$, que es la RATM que calcula la función BIT.

- De forma similar a la anterior se puede probar que $\mathcal{F}_{\text{LH}}$ también es cerrado bajo CRN. Recordando la definición exacta de este operador:

$$\begin{aligned} f(0,\vec{y}) &= g(\vec{y}) \\ f(s_0(x),\vec{y}) &= s_{h_0(x,\vec{y})}(f(x,\vec{y})), \quad \text{si } x \neq 0 \\ f(s_1(x),\vec{y}) &= s_{h_1(x,\vec{y})}(f(x,\vec{y})). \end{aligned}$$

donde se supone que los bitgrafos de las funciones $g$, $h_0$ y $h_1$ pueden ser computados por las máquinas de Turing alternantes $M_g$, $M_{h_0}$ y $M_{h_1}$ respectivamente, todas ellas de tiempo logarítmico. Una ATM $M_f$ que calcule la coordenada $i$-ésima de $f(x,\vec{y})$ podría darse mediante una máquina que ejecutara los siguientes pasos:

```
if i = |x| then
   return M_g(x)
if i < |x| then
    if M_bit(|x| - i - 1, x) = 0 then
        return M_{h_0}(|x| - i - 1, x)
    if M_bit(|x| - i - 1, x) = 1 then
        return M_{h_1}(|x| - i - 1, x)
else
    return B
```

De nuevo, cada uno de los pasos descritos pueden ejecutarse en tiempo logarítmico y, por lo tanto, $\mathcal{F}_{\text{LH}}$ es cerrada bajo CRN.

Con esto queda probado que todos los elementos de $A_0$ están en $\mathcal{F}_{\text{LH}}$ restando abordar el sentido contrario $\mathcal{F}_{\text{LH}} \subseteq A_0$. Aunque en un primer intento es razonable utilizar la codificación descrita a lo largo del apartado 4.2 para aproximarse a esta afirmación, pronto se choca con un hecho que impide esta estrategia. Si tenemos en cuenta que con dicha codificación una máquina $M$ puede tener $\mathcal{O}(\log n)$ configuraciones y a su vez, el tamaño de éstas también pueden ser $\mathcal{O}(\log n)$, se hace evidente que se requerirá números enteros de tamaño $\mathcal{O}(\log^2 n)$ para representar las posibles secuencias de configuraciones de $M$. Esto representa un problema a la hora de lograr el objetivo marcado ya que la cuantificación sobre valores de este orden no es fuertemente acotada.

En pos de lograr codificar una secuencia mediante enteros que estén dentro del alcance de la cuantificación fuertemente acotada, se puede pensar en la posibilidad de codificar secuencias de "instrucciones" en vez de secuencias de configuraciones. Si como "instrucciones" tomamos los elementos de la relación de transición $\Delta$ de $M$, las secuencias de éstas podrían ser codificadas por enteros de tamaño $n^{\mathcal{O}(1)}$, los cuales sí son cubiertos por la cuantificación fuertemente acotada. Una vez conseguida esta codificación, sólo restará demostrar que se pueden definir funciones y predicados en $A_0$ capaces de reconocer

cuando una secuencia de instrucciones se corresponde con una computación correcta.

Para describir la nueva codificación formalmente, supongamos que $M = (Q, \Sigma, \Gamma, \Delta, q_0, k+2, \ell)$ es una $\Sigma_m$ – RATM que se ejecuta en tiempo $c \cdot |n|$, con $n = |x|$. Se asume, en lo que resta de prueba, que $c = 1$ y $\Sigma = \{0, 1\}$, para simplificar la notación y mejorar la claridad de la misma. Con esto, se define una instrucción de M como un elemento de la relación de transición $\Delta$ de la forma:

$$(q, a_1, \ldots, a_{k+2}, q', b_1, \ldots, b_{k+2}, d_1, \ldots, d_{k+2}) \tag{1}$$

donde:

- $q$ es el estado actual de M.
- $a_1, \ldots, a_{k+2} \in (\Gamma \cup \{b\})$ son los símbolos que son leídos actualmente en las $k$ cintas de trabajo y las cintas de consulta y repuesta de M.
- $q'$ se refiere al siguiente estado de la máquina.
- $b_{k+2}, d_1, \ldots, d_{k+2} \in (\Gamma \cup \{b\})$ son los símbolos que serán escritos tanto en las $k$ cintas de trabajo como en las cintas de consulta y respuesta de la máquina.
- $d_i \in \{-1, 0, 1\}$, para todo $i$ tal que $1 \leq i \leq k+2$, representará la dirección del movimiento de las cabezas de cada una de las $k+2$ cintas.

Utilizando la codificación descrita en 4.2.2 sobre secuencias de instrucciones de una computación de M, se puede probar que los enteros resultantes de codificar éstas están acotados por $|x|^{\mathcal{O}(1)}$. Esto es así ya que, por una parte, cada instrucción es una tupla de longitud $3k+8$, cuya codificación va a estar acotada por $c_1(3k+8)$, con $c_1 > 0$ una constante. Pero, como M es de tiempo logarítmico, una secuencia de instrucciones de una computación suya tendrá, a lo sumo, $c_2 \log(|x|)$ elementos, con $c_2 > 0$ constante, y, por tanto, es posible codificar dicha secuencia en un entero de longitud menor que $c \log(|x|)$, donde $c$ es una constante mayor que cero.

Entonces, una máquina M acepta una entrada $x$ si y sólo si se cumple el predicado:

$$(\exists y_1 \leq |x|^d)(\forall y_2 \leq |x|^d)(\exists y_3 \leq |x|^d)\cdots(Q y_m \leq |x|^d)\Theta(x, y_1, \ldots, y_m),$$

donde Q será $\forall$ si $m$ es par ó $\exists$ si $m$ es impar y el predicado $\Theta(x, y_1, \ldots, y_m)$ es equivalente al siguiente:

$$P_1(x, y_1, \ldots, y_m) \wedge P_2(x, y_1, \ldots, y_m) \rightarrow P_3(x, y_1, \ldots, y_m)$$

en el que los predicados $P_i$ representan:

$P_1$: Para $i = 1, 2, \ldots, m$, cada $y_i$ codifica una secuencia de instrucciones de M donde los estados actuales de $y_i$ son existenciales o universales dependiendo de si $i$ es impar o par, respectivamente.

$P_2$: La secuencia de instrucciones codificada por $y_1, \ldots, y_m$ determina una correcta computación de M.

$P_3$: La secuencia de instrucciones codificada por $y_1, \ldots, y_m$ es de aceptación.

Para calcular los predicados $P_1$ y $P_3$ en $A_0$ basta actuar de manera similar a la demostración del lema 4.2.1. Usando las funciones $\tau$ se puede decodificar cada elemento de una instrucción $y_i$, con $1 \leq i \leq n$, de manera que se puede comprobar si cada uno de ellos es de la forma adecuada. Es decir, si suponemos que $y_i$ es de la forma mostrada en (1), se tiene que $P_1$ se cumple si y sólo si para cada $y_i$ ocurre:

- $q$ es universal si $i$ es par y existencial $i$ es impar.
- Si $q$ es universal entonces $q'$ es existencial y viceversa.
- Para cada $j \in \{1, \ldots, k+2\}$ se cumple que $a_j, b_j \in (\Gamma \cup \{B\}$ y $d_j \in \{-1, 0, 1\}$.

Para el caso de $P_3$, éste se cumple si y sólo si el estado de la instrucción $y_m$ es $q_A$.

En cuanto a $P_2$, es necesario un análisis más exhaustivo. Si $y_1, \ldots, y_m$ codifica una secuencia $s$ de $m \cdot \log n$ instrucciones de la máquina M, entonces $s$ se corresponderá con una computación correcta de M siempre y cuando se cumplan:

1. El estado de la primera instrucción debe ser $q_0$, el de la última debe ser a $q_A$ y, para todo $r$ tal que $0 \leq r \leq m \cdot (\log(n) - 1)$, el nuevo estado en la $r$-ésima instrucción será el estado actual de la $r+1$-ésima instrucción.
2. Para todas las celdas de cualquier cinta, y para todo $r < m \cdot \log n$, si la $r$-ésima instrucción es de la forma:

   $$(q, a_1, \ldots, a_{k+2}, q', b_1, \ldots, b_{k+2}, d_1, \ldots, d_{k+2})$$

   se tendrá que para $j$ cumpliendo $1 \leq j \leq k+2$, $a_j$ es el símbolo que estaba escrito en la $j$-ésima cinta de trabajo la última vez que la cabeza de la misma visitó la posición $p_j$, donde $p_j$ será la posición actual de la cabeza de la $j$-ésima cinta de trabajo siempre y cuando esta posición haya sido visitada previamente ó $a_j = B$ en caso contrario. Además hay que añadir el caso en que $q$ se corresponda con el estado de consulta $q_I$, donde entonces $b_{k+2} = \textsc{bit}(i, x)$, donde $i$ es el contenido actual de la cinta de consulta.

Aunque para el punto 1 no es difícil desarrollar una función en $A_0$ que lo simule, el punto 2 puede resultar no tan natural. En este caso se introduce la función bitsuma fuertemente acotada SBBITSUM tal que:

$$\textsc{sbbitsum}(x, y) = \begin{cases} \sum_{i<|y|} \textsc{bit}(i, y) & \text{si } y \leq |x| \\ |x| + 1 & \text{en caso contrario.} \end{cases}$$

Esta función resulta ser computable en tiempo $\log^2(n)$ y espacio $\log\log(n)$, lo que permite utilizar el lema 4.3.1 para ver que SBBITSUM $\in A_0$. Finalmente, esto lleva a poder determinar, dados $i_0$, $i_1$ y $j$ si la cabeza de la cinta $j$ en la instrucción $i_0$ apunta a la misma celda que en la instrucción $i_1$ en la ejecución de $M$ sobre una entrada $x$. Con esto, se ha construido en $A_0$ una codificación y una serie de predicados mediante los cuales se puede determinar cuando una secuencia de instrucciones se corresponde con una computación de aceptación correcta de $M$ y, por tanto, resolver que $\mathcal{F}\text{LH} \subseteq A_0$. □

# 5

# OTRAS CLASES DE COMPLEJIDAD.

Las herramientas y técnicas desarrolladas en el capítulo anterior para demostrar la equivalencia entre $A_0$ y $\mathcal{F}_{\text{LHC}}$ son susceptibles de ser aplicadas de manera similar en la búsqueda de resultados parecidos sobre otras clases de complejidad. Este proceso implica desarrollar predicados del tipo de los NEXT ó INICIAL anteriores para conseguir simular la computación de una máquina en una clase concreta por un álgebra de funciones dado.

## 5.1 TIEMPO POLINOMIAL (PTIME).

Salvo alguna alusión anterior, parece generalmente aceptado que fueron Alan Cobham y Jack Edmonds los primeros que hacen referencia a la clase PTIME como forma de determinar qué algoritmos son verdaderamente eficientes. De hecho, la caracterización de $\mathcal{F}_{\text{PTIME}}$ que se da a continuación (sobre enteros) fue dada para palabras de un alfabeto finito por el mismo Cobham en [4]. Será necesario para ello introducir un nuevo operador recursivo:

**Definición 5.1.1.** *Una función* $f$ *es definida por* recursión acotada sobre la notación *(en adelante* BRN*) mediante las funciones* $g$, $h_0$, $h_1$ *y* $k$ *si*

$$\left.\begin{array}{l} f(0,\vec{y}) = g(\vec{y}) \\ f(s_0(x),\vec{y}) = h_0(x,\vec{y},f(x,\vec{y})), \\ f(s_1(x),\vec{y}) = h_1(x,\vec{y},f(x,\vec{y})) \end{array}\right\} \text{si } x \neq 0$$

*con* $f(x,\vec{y}) \leq k(x,\vec{y})$ *para todo* $x$, $\vec{y}$.

Utilizando la recursión acotada sobre la notación como nuevo operador junto con algunos de los elementos dados hasta ahora se puede presentar un álgebra de funciones que determine $\mathcal{F}_{\text{PTIME}}$:

**Teorema 5.1.2.** $\mathcal{F}_{\text{PTIME}} = [0, I, s_0, s_1, \#; \text{COMP}, \text{BRN}]$.

**Demostración.** Por comodidad, a lo largo de esta prueba se va notar como $\mathcal{F}$ al álgebra $[0, I, s_0, s_1, \#; \text{COMP}, \text{BRN}]$. En primer lugar se ataca la demostración de la inclusión de izquierda a derecha, $\mathcal{F}_{\text{PTIME}} \subseteq \mathcal{F}$.

Sea $M$ una máquina de Turing determinista con entrada $x$ que corre en tiempo polinómico $p(|x|)$. El recorrido a seguir en esta parte de la prueba es prácticamente calcado a lo realizado en las secciones 4.1 y 4.2 del capítulo anterior. Se trata pues de conseguir un puñado de elementos de $\mathcal{F}$ que permitan definir las funciones $inicial_M(x)$ y $next_M(x)$ que calculan la configuración inicial de $M$ y la configuración siguiente de $M$ sobre una entrada $x$, respectivamente. Con esto, y haciendo uso de BRN, es posible dar una función $Run_M(x)$ que calcula la salida de la computación de $M$ sobre $x$.

Será de ayuda en este proceso tener a mano las siguientes funciones:

- La función $\textsc{mod2}(x)$ que, recordemos, devuelve el último dígito de la representación binaria de $x$. Se define en $\mathcal{F}$ mediante $\textsc{brn}$ como:

$$\left.\begin{array}{l}\textsc{mod2}(0) = 0\\ \textsc{mod2}(s_0(x)) = 0,\\ \textsc{mod2}(s_1(x)) = 1\end{array}\right\} \text{si } x \neq 0$$

  con $\textsc{mod2}(x) \leq x$.

- La función $\textsc{tr}(x)$ devuelve el entero resultante de truncar la última cifra de $x$ en binario. Utilizando $\textsc{brn}$ se obtiene de la siguiente forma:

$$\left.\begin{array}{l}\textsc{tr}(0) = 0\\ \textsc{tr}(s_0(x)) = x,\\ \textsc{tr}(s_1(x)) = x\end{array}\right\} \text{si } x \neq 0$$

  cumpliendo que $\textsc{tr}(x) \leq x$.

- La función $msp(x,y)$ que no es más que un refinamiento de la función $\textsc{msp}(x,y)$ que ya se dio en el capítulo anterior. Más concretamente, $msp(x,y) = \lfloor x/2^{|y|} \rfloor = \textsc{msp}(x,|y|)$, y se puede dar de nuevo usando $\textsc{brn}$:

$$\left.\begin{array}{l}msp(x,0) = x\\ msp(x,s_0(y)) = \textsc{tr}(msp(x,y)),\\ msp(x,s_1(y)) = \textsc{tr}(msp(x,y))\end{array}\right\} \text{si } x \neq 0$$

  con $msp(x,y) \leq x$ de nuevo.

- De forma similar a $msp(x,y)$ se puede definir una función $lsp(x,y)$, la cual cumple $lsp(x,y) = x \pmod{2^{|y|}} = \textsc{lsp}(x,|y|)$ y se puede obtener mediante:

$$lsp(x,y) = msp(rev(msp(rev(s_1(x)), msp(x,y))), 1).$$

  Nótese que $rev$ no ha sido dada aún en $\mathcal{F}$, pero no es difícil obtenerla sin necesidad de hacer uso de $lsp$, basta con construir la función usando $\textsc{brn}$.

- La función longitud $|x|$ se puede describir en $\mathcal{F}$ si previamente se hace lo propio con la función $s(x) = x + 1$ utilizando $\textsc{brn}$:

$$\left.\begin{array}{l}s(0) = s_1(0)\\ s(s_0(x)) = I_1^2(s_1(x), s(x)),\\ s(s_1(x)) = s_0(I_2^2(x, s(x)))\end{array}\right\} \text{si } x \neq 0$$

  con $s(x) \leq s_1(x)$. Por ejemplo, calcular $s(13)$ utilizando su representación binaria no es más que encontrar $s(1101) = s_0(I_2^2(110, I_1^2(s_1(11), s(11)))) = 1110$ y así $s(13) = 14$. Con esto, es fácil comprobar que la función longitud $|x|$ está en $\mathcal{F}$:

$$\begin{cases}|0| = 0\\ |s_i(x)| = s(|x|) \quad \text{con } |x| \leq x.\end{cases}$$

- Para finalizar, se da la función $\mathtt{bit}(x, y)$ que calcula el bit en la posición $|x|$ de $y$, es decir, $\textsc{bit}(|x|, y)$. En este caso es simple encontrarla su representación en $\mathcal{F}$:
$$\mathtt{bit}(x, y) = \textsc{mod2}(\mathtt{msp}(x, y)).$$

Con esto, tan sólo hay que seguir al pie de la letra lo realizado en las secciones del capítulo 4 anteriormente señaladas para encontrar en $\mathcal{F}$ las funciones signo, condicional, pairing y proyecciones junto con los cuantificadores del tipo es-parte-de, siempre teniendo en cuenta que se deben sustituir convenientemente MSP, LSP y BIT por las nuevas $\mathtt{msp}$, $\mathtt{lsp}$ y $\mathtt{bit}$, así como adaptar cada CRN a una BRN y sus cotas correspondientes. Por ejemplo, las proyecciones $\pi_1$ y $\pi_2$ dadas para decodificar la función pairing $\tau$ viene definida por:

$$\begin{aligned} \pi_1 &= \mathtt{msp}(x, h(x)) \\ \pi_2 &= \mathtt{lsp}(x, h(x)) \end{aligned}$$

donde $h(x)$ calcula el entero cuya representación binaria está compuesta sólo por unos y tiene una longitud igual a la mitad que $|x|$, es decir: $|h(x)| = \lfloor \frac{|x|}{2} \rfloor$. Concretamente, $h(x)$ puede darse en $\mathcal{F}$ como $h(x) = \mathtt{rev}(g(x, x))$ donde $g(x, y)$ queda definida por BRN de la siguiente forma:

$$\begin{cases} g(0, y) = 0 \\ g(s_i(x), y) = s_{\mathtt{bit}(x*x,\mathtt{ones}(y))}(g(x, y)) \quad \text{con } g(x, y) \leq |y|. \end{cases}$$

Dar en $\mathcal{F}$ todas las funciones anteriores tiene como único objetivo demostrar que se puede obtener en dicha álgebra las funciones $\mathtt{inicial}_M(x)$ y $\mathtt{next}_M(x)$ que se argumentan de manera análoga a las $\textsc{inicial}_M(x)$ y $\textsc{next}_M(x)$ dadas en 4.2.

Con tan sólo avanzar un poco más en este razonamiento es posible simular una computación de $M$ en $\mathcal{F}$. Para ello será necesario notar que mediante la composición de $0$, $s_0$, $s_1$ y $\#$ es posible definir una función $k \in \mathcal{F}$ que satisfaga $p(|x|) \leq |k(x)|$ y así, utilizando una vez más BRN, alcanzar la función:

$$\begin{aligned} \mathtt{Run}_M(x, 0) &= \mathtt{inicial}(x) \\ \mathtt{Run}_M(x, s_i(y)) &= \mathtt{next}_M(x, \mathtt{Run}_M(x, y)). \end{aligned}$$

De esta forma, el valor resultante de una computación de $M$ sobre $x$ puede deducirse de $\mathtt{Run}_M(x, k(x))$ empleando $\pi_1$ y $\pi_2$, concluyendo así que $\mathcal{F}_{\textsc{ptime}} \subseteq \mathcal{F}$.

Para el sentido contrario, $\mathcal{F} \subseteq \mathcal{F}_{\textsc{ptime}}$, se procede por inducción sobre los elementos de $\mathcal{F}$. No es necesario detenerse en exceso en el hecho de que las funciones elementales $0$, $I$, $s_0$, $s_1$ y $\#$ son calculables por máquinas de Turing en tiempo polinómico, resultando el mismo evidente en la mayoría de los casos. Por ejemplo, para la función $\#$, sea $M_1$ una TM que calcula el $i$–ésimo bit de $x\#y = 2^{|x|\cdot|y|}$, devolviendo $1$ si $i = |x| \cdot |y|$ y $0$ en cualquier otro caso. Como el producto $|x| \cdot |y|$ puede ser calculado en tiempo $\log^2(n)$ se concluye que $\# \in \mathcal{F}_{\textsc{ptime}}$.

Probar que $\mathcal{F}$PTIME es cerrado bajo composición sigue un procedimiento totalmente análogo al realizado en el teorema 4.3.3 mientras que para BRN se puede suponer la existencia de $M_g$, $M_{h_0}$, $M_{h_1}$ y $M_k$, todas TM en tiempo polinómico que calculan las funciones $g$, $h_0$, $h_1$ y $k$ respectivamente. Así, si $f$ está definida de la forma:

$$\left.\begin{array}{l} f(0,\vec{y}) = g(\vec{y}) \\ f(s_0(x),\vec{y}) = h_0(x,\vec{y},f(x,\vec{y})), \\ f(s_1(x),\vec{y}) = h_1(x,\vec{y},f(x,\vec{y})) \end{array}\right\} \text{si } x \neq 0$$

con $f(x,\vec{y}) \leq k(x,\vec{y})$, una máquina de Turing $M_f$ que calcule la misma podría seguir el algoritmo siguiente:

```
if x = 0 then
   return M_g(y)
if x ≠ 0 then
    if M_bit(0,x) = 0 then
        return M_h0(M_TR(x),y,M_f(x−1,y))
    if M_bit(0,x) = 1 then
        return M_h1(M_TR(x),y,M_f(x−1,y))
else
    return B
```

Donde $M_{TR}$ es la máquina que calcula la función TR. Este procedimiento llamará a $M_{h_0}$ ó $M_{h_1}$ a lo sumo $n = |x|$ veces y, al ser éstas de tiempo polinómico, $M_f$ dará a lo sumo $n^{\mathcal{O}(1)}$ pasos. Se observa entonces que $\mathcal{F}$PTIME es cerrada bajo composición y recursión acotada sobre la notación, teniendo por tanto que $\mathcal{F} \subseteq \mathcal{F}$PTIME. □

## 5.2 ESPACIO LOGARÍTMICO (LOGSPACE).

La siguiente caracterización de $\mathcal{F}$LOGSPACE no es más que una versión aritmética del resultado de John C. Lind en [11] que, como en el caso del álgebra de Cobham para $\mathcal{F}$PTIME, fué desarrollado para palabras de un cierto alfabeto $\Sigma^*$. En particular, la describe como la menor clase de funciones que contiene la función característica de la igualdad, la función concatenación $*$ y es cerrada bajo los operadores de transformación explícita, recursión $\log$–acotada sobre la notación y una versión de recursión estrictamente más potente que CRN.

**Definición 5.2.1.** *Una función $f$ está definida por la funciones $g$, $h_0$, $h_1$, $k$ mediante* recursión fuertemente acotada sobre la notación *(en lo que sigue* SBRN*) si:*

$$\left.\begin{array}{l} f(0,\vec{y}) = g(\vec{y}) \\ f(s_0(x),\vec{y}) = h_0(x,\vec{y},f(x,\vec{y})), \\ f(s_1(x),\vec{y}) = h_1(x,\vec{y},f(x,\vec{y})) \end{array}\right\} \textit{si } x \neq 0$$

*con $f(x,\vec{y}) \leq |k(x,\vec{y})|$ para todo $x$, $\vec{y}$.*

**Teorema 5.2.2.** *$\mathcal{F}$LOGSPACE se corresponde con las siguientes álgebras:*

$$\begin{aligned} \mathcal{F}\text{LOGSPACE} &= [0, I, s_0, s_1, |x|, \text{BIT}, \#; \text{COMP}, \text{CRN}, \text{SBRN}] \\ &= [0, I, s_0, s_1, \text{MOD2}, msp, \#; \text{COMP}, \text{CRN}, \text{SBRN}]. \end{aligned}$$

**Demostración.** De esta prueba sólo se darán detalles sobre la primera igualdad, pudiéndose comprobar la segunda mediante las mismas técnicas. Por comodidad, se denominará temporalmente como $\mathcal{F}$ al álgebra de funciones $[0, I, s_0, s_1, |x|, \text{BIT}, \#; \text{COMP}, \text{CRN}, \text{SBRN}]$.

Para la inclusión $\mathcal{F} \subseteq \mathcal{F}_{\text{LOGSPACE}}$ pueden usarse, casi en el mismo orden, las herramientas y procedimientos de las demostraciones de los teoremas 4.3.3 y 5.1.2.

Para probar $\mathcal{F}_{\text{LOGSPACE}} \subseteq \mathcal{F}$ de nuevo se recurre a la simulación de la computación de una máquina de Turing M sobre una entrada $x$ mediante las funciones $\text{INICIAL}_M(x)$ y $\text{NEXT}_M(x)$, teniendo en cuenta esta vez que el espacio requerido por M está acotado por una función en $\mathcal{O}(\log(|x|))$. Observando que $A_0 \subseteq \mathcal{F}$ (pues $A_0$ y $\mathcal{F}$ son idénticas salvo la recursión SBRN de esta última), tan sólo es necesario recurrir al lema 4.2.1 para ver que tanto $\text{INICIAL}_M(x)$ como $\text{NEXT}_M(x)$ están en $\mathcal{F}$. Así, como es de esperar, usando SBRN puede darse la función:

$$\begin{aligned} \text{RUN}_M(x, 0) &= \text{INICIAL}_M(x) \\ \text{RUN}_M(x, s_i(y)) &= \text{NEXT}_M(x, \text{RUN}_M(x, y)) \end{aligned}$$

tal que $\text{RUN}_M(x, y) \leq |k(x)|$ donde $k(x)$ es una función en $\mathcal{F}$ que supone un límite superior adecuado para las configuraciones de M. Se concluye pues que de $\text{RUN}_M(x, k(x))$ puede calcular el resultado de una computación de M sobre $x$ y, por tanto, $\mathcal{F}_{\text{LOGSPACE}} \subseteq \mathcal{F}$. □

## 5.3 ESPACIO LINEAL (LINSPACE).

Andrzej Grzegorczyk introdujo e investigó en 1953 ([9]) una sucesión de funciones recursivas primitivas tal que cada función en ella tiene un crecimiento más complejo que la función que la precede en la sucesión. Estas funciones permiten dar una jerarquía de subclases definidas como el cierre de éstas (más ciertas funciones iniciales) bajo composición y recursión acotada, clasificando de esta manera la funciones recursivas primitivas.

**Definición 5.3.1.** *Una función* $f$ *está definida por la funciones* $g$, $h$, $k$ *mediante* recursión acotada *(notada por* BR*) si:*

$$\begin{aligned} f(0, \vec{y}) &= g(\vec{y}) \\ f(x+1, \vec{y}) &= h(x, \vec{y}, f(x, \vec{y})), \end{aligned}$$

*con* $f(x, \vec{y}) \leq k(x, \vec{y})$ *para todo* $x$, $\vec{y}$.

**Definición 5.3.2.** *Diremos que una función* $f^{(n)}(x)$ *está* definida por inducción *sobre* $n$ *si:*

$$\begin{cases} f^{(0)}(x) = x, \\ f^{(n+1)}(x) = f(f^{(n)}(x)). \end{cases}$$

Seguidamente se describen las funciones de Grzegorczyk, aunque la versión original ha sido sustituida por una totalmente equivalente que parece un poco más intuitiva a la vez que será de utilidad en resultados posteriores.

**Definición 5.3.3.** *Se definen las siguientes funciones* principales *como:*

$$\begin{aligned} f_0(x) &= s(x) = x+1 \\ f_1(x,y) &= x+y \\ f_2(x,y) &= (x+1)\cdot(y+1) \\ f_3(x) &= 2^x \\ f_{n+1}(x) &= f_n^{(x)}(1), \text{ para } n \geq 3. \end{aligned}$$

*Se denotará como* $\mathcal{E}f$ *al álgebra* $[0, I, s, \text{máx}, f; \text{COMP}, \text{BR}]$ *así como* $\mathcal{E}^n$ *se refiere a* $\mathcal{E}f_n$.

Teniendo las dos últimas definiciones, se puede dar como ejemplo $f_4(x)$:

$$f_4(x) = f_3^{(x)}(1) = \underbrace{f_3(f_3(\ldots f_3}_{x \text{ veces}}(1))) = 2^{2^{\cdot^{\cdot^{2^1}}}} \Big\}^{\text{Altura } x}.$$

Además de los dos siguientes resultados (5.3.4 y 5.3.6), Grzegorczyk también prueba en [9] que para todo $n \geq 0$ se cumple que $\mathcal{E}^n$ está estrictamente contenido en $\mathcal{E}^{n+1}$ mediante la demostración de que $f_{n+1} \notin \mathcal{E}^n$. También consigue probar un resultado similar para $\mathcal{E}_*^n$ y $\mathcal{E}_*^{n+1}$ aunque en este caso para $n \geq 2$.

El siguiente lema y su correspondiente corolario van a servir para describir algunos elementos de $\mathcal{E}^n$ que posteriormente ayudarán a caracterizar $\mathcal{F}$LINSPACE.

**Lema 5.3.4.** *Las funciones* $x \dot{-} y$, $sg(x)$, $\overline{sg}(x)$, $sg(x)\cdot y$, $\overline{sg}(x)\cdot y$ *pertenecen a* $\mathcal{E}^0$. *Si* $f$ *es una función en* $\mathcal{E}^0$ *entonces* $\sum_{i\leq x} sg(f(i))$, $\sum_{i\leq x} \overline{sg}(f(i))$, $\prod_{i\leq x} sg(f(i))$ *y* $\prod_{i\leq x} \overline{sg}(f(i))$ *también están en* $\mathcal{E}^0$.

**Demostración.** Esta demostración no entraña dificultad alguna pudiéndose consultar en [9] (pág. 30 a la 33). A modo de ejemplo se dejan una prueba de cada apartado.
La función $x \dot{-} y$ se da en $\mathcal{E}^0$ utilizando recursión acotada:

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} (y+1) &= (x \dot{-} y) \dot{-} 1 \end{aligned}$$

con $x \dot{-} y \leq x$ y $x \dot{-} 1$ también dado por BR si $0 \dot{-} 1 = 0$, $(x+1) \dot{-} 1 = x$ y $x \dot{-} 1 \leq x$. Igualmente puede usarse BR para dar $g(x) = \sum_{i\leq x} sg(f(i))$:

$$\begin{aligned} g(0) &= sg(f(0)) \\ g(x+1) &= h(sg(f(x), g(x))) \end{aligned}$$

con $g(x) \leq x+1$ y la función $h(x,y)$ definida mediante recursión acotada como:

$$\begin{aligned} h(x,0) &= x+1 \\ h(x,y+1) &= x \end{aligned}$$

con $h(x,y) \leq x+1$. □

**Definición 5.3.5.** *Se dice que una función* $f$ *está definida por* minimización acotada *(notado* BMIN*) mediante la función* $g$ *si:*

$$f(x,\vec{y}) = \begin{cases} \text{mín}\{i \le x : g(i,\vec{y}) = 0\}, & \textit{si existe tal } i \\ 0, & \textit{en caso contrario.} \end{cases}$$

*Su notación viene dada por* $f(x,\vec{y}) = \mu i \le x[g(i,\vec{y}) = 0]$.

**Corolario 5.3.6.** *Para* $n \ge 0$, $\mathcal{E}^n_*$ *es cerrado bajo conectivas booleanas y cuantificación acotada. Además, se cumple que* $\mathcal{E}^n$ *es cerrada bajo minimización acotada.*

**Demostración.** El predicado $\neg P(\vec{x})$ tiene como función característica

$$\overline{sg}(c_P(\vec{x})),$$

la función característica del predicado $P(\vec{x}) \vee Q(\vec{x})$ es

$$\overline{sg}(\overline{sg}(c_P(\vec{x})) \cdot \overline{sg}(c_Q(\vec{x}))),$$

la del predicado $(\exists i \le x)R(i,\vec{y})$ es

$$sg(s(x) \dot{-} \sum_{i\le x} \overline{sg}(c_R(i,\vec{y}))),$$

y, finalmente, la función característica de $(\forall i \le x)R(i,\vec{y})$ es

$$\overline{sg}(s(x) \dot{-} \sum_{i\le x} c_R(i,\vec{y})).$$

Para definir $f(x,\vec{y}) = \mu i \le x[g(i,\vec{y}) = 0]$ se da en primer lugar la función $h$ tal que

$$h(i,\vec{y}) = \sum_{j\le i} \overline{sg}(g(j,\vec{y})),$$

la cual calcula el cardinal del conjunto $\{j \le i : g(j,\vec{y}) = 0\}$. Como $\overline{sg}(h(i,\vec{y})) = 1$ si y sólo si se cumple el predicado $(\forall j \le i)(g(j,\vec{y}) \neq 0)$ se tiene

$$\sum_{i\le x} \overline{sg}(h(i,\vec{y})) = \begin{cases} \mu i \le x[g(i,\vec{y}) = 0], & \text{si } (\exists i \le x)(g(j,\vec{y}) = 0) \\ x+1, & \text{en caso contrario.} \end{cases}$$

Por tanto, $f$ se define en $\mathcal{E}^n$ como:

$$f(x,\vec{y}) = \overline{sg}((\sum_{i\le x} \overline{sg}(h(i,\vec{y}))) \dot{-} x) \cdot \sum_{i\le x} \overline{sg}(h(i,\vec{y})).$$

□

La siguiente caracterización de $\mathcal{F}_{\text{LINSPACE}}$ se debe a Robert W. Ritchie ([14]), quien describe a la misma en base a la jerarquía de Grzegorczyk.

**Teorema 5.3.7.** $\mathcal{F}_{\text{LINSPACE}} = \mathcal{E}^2$.

**Demostración.** Como anteriormente, se demuestra por doble inclusión. La dirección de derecha a izquierda se deja al lector ya que se puede probar usando las técnicas descritas en demostraciones anteriores (4.3.3, 5.1.2) que las funciones iniciales de $\mathcal{E}^2$ son computables en LINSPACE a la vez que $\mathcal{F}_{\text{LINSPACE}}$ es cerrado bajo composición y recursión acotada.

Para probar la dirección contraria, $\mathcal{F}_{\text{LINSPACE}} \subseteq \mathcal{E}^2$, es interesante corroborar primero la siguiente afirmación:

$$[0, I, s_0, s_1, \text{MOD2}, msp; \text{COMP}, \text{CRN}] \subseteq \mathcal{E}^2.$$

Se pretende con esto, al igual que se hizo en la demostración del teorema 5.1.2, encontrar que en un álgebra con los elementos anteriores se pueden dar las funciones $inicial_M$ y $next_M$ que ayuden a describir la computación de una cierta máquina M.

Es evidente que tanto $s_0$ como $s_1$ pertenecen a $\mathcal{E}^2$ y, por el corolario 5.3.6, también $cond \in \mathcal{E}^2$ ésta es cerrada bajo cuantificación acotada. Para el resto se necesita un poco más de detalle:

- Para ver que $\text{MOD2} \in \mathcal{E}^2$ se define $\overline{sg}$ como $\overline{sg}(0) = 1$ y $\overline{sg}(s(x)) = 0$ y, usando BR se llega a:

  $$\begin{aligned} \text{MOD2}(0) &= 0 \\ \text{MOD2}(s(x)) &= \overline{sg}(\text{MOD2}(x)) \end{aligned}$$

  con $\text{MOD2}(x) \leq 1$.

- Antes de probar que $msp \in \mathcal{E}^2$ será necesario encontrar $\text{MSP}, |x| \in \mathcal{E}^2$. Para la primera, si se tiene en cuenta:

  $$\lfloor \frac{x}{2} \rfloor = \mu y \leq x[y + y = x \vee y + y + 1 = x]$$

  se puede usar recursión acotada para definir MSP:

  $$\begin{aligned} \text{MSP}(x, 0) &= x \\ \text{MSP}(x, i+1) &= \lfloor \text{MSP}(x, i)/2 \rfloor \end{aligned}$$

  con $\text{MSP}(x, y) \leq y$. De aquí se deduce también $\text{BIT}(i, x) = \text{MOD2}(\text{MSP}(x, i))$. Para $|x|$ se utiliza la función auxiliar $h$ definida por BR y $cond$:

  $$\begin{aligned} h(x, 0) &= 0 \\ h(x, i+1) &= \begin{cases} h(x, i) + 1, & \text{si } \text{BIT}(i, x) = 1 \\ h(x, i), & \text{en caso contario.} \end{cases} \end{aligned}$$

  Con $h(x, y) \leq x$. Por tanto, se puede describir $|x|$ en $\mathcal{E}^2$ como $|x| = h(ones(x), x)$ donde $ones(x) = 2^{|x|} - 1$ viene dado por:

  $$\mu y \leq s_0(x)[(\forall i \leq x)(\text{BIT}(i, y) = 1 \leftrightarrow (\exists j \leq y)(i \leq j \wedge \text{BIT}(j, x) = 1))].$$

  Ya sólo resta observar que $msp(x, y) = \text{MSP}(x, |y|)$ y, de esta forma, está en $\mathcal{E}^2$.

- Supuesto que la función f está definida por g, $h_0$, $h_1 \in \mathcal{E}^2$ mediante CRN y siendo $\mathcal{E}^2$ cerrada bajo minimización acotada (corolario 5.3.6) se define f como:

$$\begin{aligned} f(x,\vec{y}) = \mu z \leq g(\vec{y})\cdot(2x+1)+2x\Big\{ & \big[|z| = |g(\vec{y})| + |x|\big] \vee \\ & \big[\text{MSP}(z,|x|) = g(\vec{y})\big] \vee \\ & \big[(\forall i,j<|x|)[j = |x| - i - 1 \wedge \text{BIT}(j,x) = 0 \longrightarrow \\ & \quad \text{BIT}(j,z) = h_0(\text{MSP}(x,j+1),\vec{y})]\big] \vee \\ & \big[(\forall i,j<|x|)[j = |x| - i - 1 \wedge \text{BIT}(j,x) = 1 \longrightarrow \\ & \quad \text{BIT}(j,z) = h_1(\text{MSP}(x,j+1),\vec{y})]\big]\Big\} \end{aligned}$$

  que no es más que una descripción exhaustiva de CRN y así queda probado que $\mathcal{E}^2$ es cerrado bajo la misma. Conviene reseñar que la cota de la minimización es suficiente ya que $f(x,\vec{y}) \leq g(\vec{y}\cdot 2^{|x|}+2^{|x|})-1$ y, por tanto, a lo sumo $g(\vec{y})\cdot(2x+1)+2x$.

Finalizada la prueba de $[0, I, s_0, s_1, \text{MOD2}, \text{msp}; \text{COMP}, \text{CRN}] \subseteq \mathcal{E}^2$ y siguiendo la demostración del teorema 5.1.2, se tiene que las funciones $\text{inicial}_M(x)$ y $\text{next}_M(x)$ pertenecen a $\mathcal{E}^2$ donde M es una máquina de Turing multicinta de espacio lineal que calcula la función f sobre x. La computación de M sobre x puede darse en $\mathcal{E}^2$ usando recursión acotada:

$$\begin{aligned} T(x,0) &= \text{inicial}_M(x) \\ T(x,y+1) &= \text{next}_M(T(x,y)). \end{aligned}$$

La cota para $T(x,y)$ viene del hecho de que al ser M de espacio lineal existirá una constante c tal que $|T(x,y)| \leq c\cdot|x|$ y, por tanto, $T(x,y) \leq (x+1)^c+1$. Queda probado pues que $\mathcal{F}_{\text{LINSPACE}} \subseteq \mathcal{E}^2$. □

**Corolario 5.3.8.** LINSPACE $= \mathcal{E}^2_*$.

## 5.4 JERARQUÍA DE TIEMPO LINEAL (LTH).

Además de la jerarquía $\mathcal{E}^n$, Grzegorczyk también presenta en [9] las siguientes clases de funciones que tienen como nuevo operador la minimización acotada.

**Definición 5.4.1.** *Para todo* $n \leq 0$ *se define el álgebra de funciones* $\mathcal{M}^n$ *como:*

$$\mathcal{M}^n = [0, I, s, \text{máx}, f_n; \text{COMP}, \text{BMIN}].$$

Grzegorczyk afirma que para cada $n \geq 3$, $\mathcal{E}^3 = \mathcal{M}^3$, lo cual fue posteriormente probado por K. Harrow.

**Definición 5.4.2.** *Se define el* álgebra de funciones rudimentarias *(que llamaremos* RF*) como:*

$$\text{RF} = [0, I, s, +, \times; \text{COMP}, \text{BMIN}].$$

En [15] Julia Robinson define la cuantificación acotada y la suma usando para ello la función sucesor y el producto, por lo que podemos considerar que $\mathcal{M}^2 = \text{RF}$.

**Definición 5.4.3.** *El* Lenguaje de Primer Orden de la Aritmética *consta de los símbolos no lógicos* $0$, $s$, $+$, $\cdot$ *y* $\leq$. *Los términos de este lenguaje se define por inducción como:*

- $0$ *es un término del lenguaje.*
- *Si* $t$ *y* $t'$ *son términos, entonces* $s(t)$, $t + t'$ *y* $t \cdot t'$ *son términos también.*

*Las* fórmulas atómicas *serán de la forma* $t = t'$ *y* $t \leq t'$, *donde* $t$ *y* $t'$ *son términos.*

**Definición 5.4.4.** *El* conjunto de fórmulas acotadas, $\Delta_0$, *de define por inducción como:*

- *Si* $\phi$ *es una fórmula atómica, entonces* $\phi \in \Delta_0$.
- *Si* $\phi, \theta \in \Delta_0$, *entonces* $\neg\phi$, $\phi \wedge \theta$ *y* $\phi \vee \theta$ *pertenecen a* $\Delta_0$.
- *Si* $\phi \in \Delta_0$ *y* $t$ *es un término, entonces* $(\exists x \leq t)\phi(x, t)$ *y* $(\forall x \leq t)\phi(x, t)$ *están en* $\Delta_0$.

*Finalmente, se dice que una relación de aridad* $k$, $R \subseteq \mathbb{N}^k$ *pertenece a* $\Delta_0^N$ *si existe una fórmula* $\phi \in \Delta_0$ *para la cual* $R = \{(a_1, \ldots, a_k) : N \vDash \phi(a_1 \ldots, a_k)\}$.

Algunos conceptos de las últimas definiciones podrían necesitar ser explicados con mayor detalle, para ello se puede consultar cualquier manual de Lógica Matemática (por ejemplo [10]).

**Definición 5.4.5.** *Se dice que un predicado* $R \subseteq \mathbb{N}^k$ *es* aritmético constructivo *(que escribiremos* $R \in \text{CA}$*) si existe una fórmula* $\phi(\vec{x}) \in \Delta_0$ *tal que* $R(a_1, \ldots, a_n)$ *se cumple si y sólo si* $\mathbb{N} \vDash \phi(a_1, \ldots, a_n)$.
*Se tiene así que una función* $f(\vec{x}) \in \mathcal{G}\text{CA}$ *si su bitgrafo* $B_f \in \text{CA}$ *y* $f$ *es de crecimiento lineal.*

Esta definición nos va a permitir dar la siguiente caracterización de $\mathcal{M}^2$:

**Proposición 5.4.6.** $\mathcal{M}^2_* = \text{CA}$ *y* $\mathcal{M}^2 = \mathcal{G}\text{CA}$.

La demostración de esta proposición se basa de nuevo en el hecho de que la suma puede ser definida en base a la función sucesor y el producto ([15]), es decir, si $a$, $b$ y $c$ son enteros, entonces se tiene que $a + b = c$ si y sólo se satisface la siguiente fórmula:

$$s(a \cdot c) \cdot s(b \cdot c) = s(s(c \cdot c) \cdot s(a \cdot b)).$$

Recapitulando lo visto hasta ahora en esta sección, se ha introducido RF y CA, y se ha observado que $\mathcal{M}^2 = \text{RF} = \mathcal{G}\text{CA}$. Esto último hace patente la estrecha relación existente entre la minimización acotada y la cuantificación acotada.

Para llegar a nuestro objetivo, o sea, dar un álgebra que caracterice la jerarquía de tiempo lineal, vamos a probar que LTH coincide con CA, para lo cual es necesario introducir el siguiente resultado.

**Proposición 5.4.7.** *La relación* $G(x,y,z)$ *del grafo de la exponencial,* $x^y = z$*, está en la aritmética constructiva.*

**Demostración.** La prueba se basa en dar un algoritmo cuyo cálculo pueda codificarse en $\Delta_0$ de forma que todos los cuantificadores estén acotados por un polinomio en $z$. Si suponemos que $x,y > 1$ con $n = |y|$ e $y = \sum_{i<n} y_i \cdot 2^i$ la representación binaria de $y$, un algoritmo que calcule $x^y = z$ puede ser el siguiente:

```
z = 1
for i = 1 to n
    if y_{n-i} = 0 then
        z = z^2
    else
        z = z^2 · x
```

Este algoritmo se basa en que es posible calcular $x^y = z$ aplicando de forma reiterada que $x^{2y} = (x^y)^2$ y $x^{2y+1} = (x^y)^2 \cdot x$.

Para codificar el cálculo del algoritmo anterior encontramos los enteros $a_i$ y $b_i$ que, para $i \in \{0,\dots n\}$, se describen como:

- $a_i = \text{MSP}(y, |y| - i)$, es decir, los $i$ bits más a la izquierda de la representación binaria de $y$.
- $b_i = x^{a_i}$, así, $b_i$ será igual al valor de $z$ en la $i$-ésima repetición del bucle for del algoritmo.

Si excluimos los casos triviales $x = 0$ e $y = 1$, se tiene que $x^y = z$ si y sólo si existen sendas secuencias, $(a_0,\dots,a_n)$ y $(b_0,\dots,b_n)$ cumpliendo los siguientes predicados:

- $a_0 = 0 \wedge b_0 = 1$.
- $a_n = y \wedge b_n = z$.
- $(\forall i < n)(a_{i+1} \in \{2 \cdot a_i, 2 \cdot a_i + 1\})$.
- $(\forall i < n)((\text{LSP}(a_{i+1}, 1) = 0 \to b_{i+1} = b_i^2) \wedge$
  $(\text{LSP}(a_{i+1}, 1) = 1 \to b_{i+1} = b_i^2 \cdot x))$.

Con esto, para demostrar que el grafo de la exponencial está en $\Delta_0$ sólo quedaría probar que los cuantificadores de los predicados anteriores se pueden acotar por un polinomio en $z$. Para ello, se van a encontrar una serie de enteros primos entre si, $m_0 < m_1 < \dots < m_n$ que satisfagan que $a_i, b_i < m_i$ para $0 \le i \le n$. De esta forma, aplicando el Teorema Chino del Resto se obtendría $M = \prod_{i \le n} m_i$ y dos enteros $A, B$ únicos, menores estrictos que $M$, tal que, para cada $i$:

$$\begin{aligned} A &\equiv a_i \mod m_i \\ B &\equiv b_i \mod m_i. \end{aligned}$$

Si tomamos cada $m_i$ como potencias de un primo distinto, y $m_{i+1}$ la menor potencia de un primo que divide a $M$ y es mayor que $M_i$, podemos determinar en $\Delta_0$ cada $m_i$ de $M$. Formalmente, a esto se llega mediante los siguientes predicados:

- $\textsc{primo}(p)$, que expresa que $p$ es primo:

$$\textsc{primo}(p) \Leftrightarrow 2 \leq p \wedge (\forall p < q)(q|p \rightarrow q = 1).$$

- $\textsc{mpp}(m, M)$, que expresa que $m$ es la mayor potencia de un primo concreto que divide a $M$:

$$\begin{aligned}\textsc{mpp}(m, M) \Leftrightarrow m|M \wedge (\exists p \leq m)(\textsc{primo}(p) \wedge p|m \wedge \\ (\forall q < m)(q|m \rightarrow q \in \{1, p\}) \wedge p \cdot m \nmid M).\end{aligned}$$

- $\textsc{i}(m, M)$, que expresa que $m = m_0$ es la mayor potencia de un primo que divide a $M$ y, a su vez, no existe ningún otro primo cuya potencia divida a $M$ y sea menor que $m$:

$$\begin{aligned}\textsc{i}(m, M) \Leftrightarrow (m = 1 \wedge M \in \{0, 1\}) \vee \\ (\textsc{mpp}(m, M) \wedge (\forall m' \leq M)(\textsc{mpp}(m', M) \rightarrow m \leq m')).\end{aligned}$$

- $\textsc{n}(m, m', M)$, que expresa que $m' = m_{i+1}$ es la siguiente mayor potencia de un primo que divide a $M$ después de $m = m_i$:

$$\begin{aligned}\textsc{n}(m, m', M) \Leftrightarrow \textsc{mpp}(m, M) \wedge \textsc{mpp}(m', M) \wedge m < m' \wedge \\ (\forall m'' < m')(\textsc{mpp}(m'', M) \rightarrow m'' \leq m).\end{aligned}$$

- $\textsc{f}(m, M)$ que expresa que $m = m_n$ es la mayor potencia de un primo que divide a $M$:

$$\begin{aligned}\textsc{f}(m, M) \Leftrightarrow (m = 1 \wedge M \in \{0, 1\}) \vee \\ (\textsc{mpp}(m, M) \wedge (\forall m' \leq M)(\textsc{mpp}(m', M) \rightarrow m' \leq m)).\end{aligned}$$

Supongamos ahora $x$ e $y$ distintos de $0, 1$ y, por tanto, que $a_i < 2^i$ y $b_i \leq x^{2^i}$ para $i \leq n$. En estas condiciones, para definir una secuencia incremental $m_0, \ldots, m_n$ de potencias de primos distintos entre si comenzamos tomando $m_0 = 2$. De esta forma, dados $m_0, \ldots, m_{k-1}$ se escoge $m_k = p^\alpha$, donde $p$ es el menor número primo que no divide a $\prod_{i<k} m_i$ y $\alpha$ es el menor entero tal que $p^\alpha > x^{2^k}$. Del Teorema de los números primos se deduce que $p = \mathcal{O}(k \cdot \ln k) < k^2 \leq x^{2^k}$ y, por la elección que hemos hecho de $\alpha$, que $p^{\alpha-1} < x^{2^k}$, teniendo:

$$p^\alpha = p \cdot p^{\alpha-1} < x^{2^k} \cdot x^{2^k} \leq x^{2^{k+1}}.$$

Finalmente, se prueba mediante inducción que para todo $k > 0$ se cumple que $\prod_{i<k} m_i \leq x^{2^{k+1}}$ y, por tanto, podemos acotar $M$ como sigue:

$$M \leq \prod_{i \leq n} m_i \leq x^{2^{n+2}} = (x^{2^{n-1}})^8 \leq z^8.$$

Obtenida la cota $z^8$ para los cuantificadores de la codificación, concluimos que la relación $x^y = z$ pertenece a $\Delta_0$. □

**Corolario 5.4.8.** *El álgebra de funciones* $[0, I, s_0, s_1, |x|, \text{BIT}; \text{COMP}, \text{CRN}]$ *está contenido en* $\mathcal{M}^2$.

**Demostración.** Las funciones $s_0$, $s_1$, $|x|$ y BIT pueden definirse en $\mathcal{M}^2$ como:

- $s_0(x) = x + x$.
- $s_1(x) = x + x + 1$.
- $|x| = \mu y \leq x[(\exists z \leq 2 \cdot x)(2^y = z \wedge x < z \wedge \lfloor \frac{z}{2} \rfloor \leq x)]$.
- $\text{BIT}(i, x) = \mu y \leq 1[(\exists u, v \leq x)(|u| = i + 1 \wedge v = 2^{i+1} \wedge v|(x - u))]$.

Usando las funciones anteriores y minimización acotada no es difícil probar que $\mathcal{M}^2$ es cerrada para CRN. □

Un último lema antes de abordar la prueba de LTH = CA es una generalización de resultado 4.3.1.

**Lema 5.4.9.** *Para todo* $k, m > 1$ *se cumple que:*

$$\text{NTIMESPACE}(n^k, n^{1-1/m}) \subseteq \text{LTH}.$$

*Más aún:*

$$\text{NSPACE}(\mathcal{O}(\log(n))) \subseteq \text{LTH}.$$

No vamos a dar la demostración de este lema dada su similitud con las pruebas de los resultados 4.3.2 y 4.3.1.

**Teorema 5.4.10.** LTH = CA.

**Demostración.** Comenzamos con la dirección LTH ⊆ CA. El álgebra referida en el lema 5.4.8 es exactamente el mismo que utilizamos en el teorema 5.1.2, por lo que se puede probar de forma similar que las funciones $inicial_M$ y $next_M$ están en CA e, igualmente, se puede simular la computación de una máquina de Turing en tiempo lineal dentro de CA.

En cuanto a la dirección contraria, CA ⊆ LTH, se puede probar mediante inducción sobre la longitud de las fórmulas de $\Delta_0$. Un esquema de la prueba sería demostrar en primer lugar que la suma, el predicado orden y el producto son computables en LOGSPACE para después probar que $\mathcal{F}$LOGSPACE es cerrado bajo composición. Esto puede realizarse utilizando para ello técnicas muy parecidas a las dadas a lo largo del presente trabajo. Con esto, por el lema 5.4.9 se tiene que las fórmulas atómicas de $\Delta_0$ definen relaciones en LTH.

Por último, al igual que en la prueba del teorema 4.3.3, los cuantificadores acotados pueden darse mediante derivaciones existenciales y universales de una máquina de Turing alternante. □

De este teorema y de la proposición 5.4.6 se obtiene la caracterización de la jerarquía de tiempo lineal que andábamos buscando.

**Corolario 5.4.11.** $\mathcal{M}^2_* = \text{LTH}$ *y* $\mathcal{M}^2 = \mathcal{F}\text{LTH}$.

Como último resultado del presente trabajo vamos a reseñar una caracterización de la jerarquía de tiempo polinomial que puede probarse aplicando desarrollos parecidos a los aparecidos a lo largo de este capítulo.

**Teorema 5.4.12.**

$$\begin{aligned}\mathcal{F}\text{PH} &= [0, \text{I}, \text{s}, +, \dot{-}, \times, \# \,; \text{COMP}, \text{BMIN}] \\ &= [0, \text{I}, \text{s}, +, \dot{-}, \times, \# \,; \text{COMP}, \text{BRN}, \text{BMIN}].\end{aligned}$$

# BIBLIOGRAFÍA


[1] S. R. Buss. "The Boolean Formula Value Problem is in ALOGTIME". En: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: ACM, 1987, págs. 123-131. URL: http://doi.acm.org/10.1145/28395.28409.

[2] Ashok K. Chandra, Dexter C. Kozen y Larry J. Stockmeyer. "Alternation". En: *J. ACM* 28.1 (ene. de 1981), págs. 114-133. URL: http://doi.acm.org/10.1145/322234.322243.

[3] Peter Clote. "Computation models and function algebras". En: *Lecture Notes in Computer Science*. Springer Science Business Media, 1995, págs. 98-130. URL: http://dx.doi.org/10.1007/3-540-60178-3_81.

[4] Alan Cobham. "The Intrinsic Computational Difficulty of Functions". En: *Logic, Methodology and Philosophy of Science: Proceedings of the 1964 International Congress (Studies in Logic and the Foundations of Mathematics)* (1965), págs. 24-30. URL: https://www.cs.toronto.edu/~sacook/homepage/cobham_intrinsic.pdf.

[5] Stephen Cook y Phuong Nguyen. *Logical Foundations of Proof Complexity*. Cambridge University Press, 11 de ene. de 2010. URL: http://www.ebook.de/de/product/9650549/stephen_cook_logical_foundations_of_proof_complexity.html.

[6] Martin Davis, Ron Sigal y Elaine J. Weyuker. *Computability, Complexity and Languages*. Elsevier Science & Technology, 11 de mar. de 1994. 609 Seiten. URL: http://www.ebook.de/de/product/3237118/martin_davis_ron_sigal_elaine_j_weyuker_computability_complexity_and_languages.html.

[7] Ding-Zhu Du y Ker-I Ko. *Theory of Computational Complexity*. 2.ª ed. Wiley Series in Discrete Mathematics and Optimization. Wiley, 2014.

[8] Edward R. Griffor. *Handbook of Computability Theory*. ELSEVIER LTD, 11 de oct. de 1999. 724 Seiten. URL: http://www.ebook.de/de/product/5491322/e_r_griffot_edward_r_griffor_handbook_of_computability_theory.html.

[9] Andrzej Grzegorczyk. "Some classes of recursive Functions". En: *Rozprawy matematyczne* 4 (1953), págs. 1-45. URL: http://matwbn.icm.edu.pl/ksiazki/rm/rm04/rm0401.pdf.

[10] *Lógica Matemática*. https://www.cs.us.es/cursos/lm/apuntes-2012.pdf: Depto. Ciencias de la Computación e Inteligencia Artificial, Universidad de Sevilla, 2011.

[11] John C. Lind. "Computing in logarithmic space". En: *Technical Report Project MAC Technical Memorandum* 52 (sep. de 1974). URL: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TM-052.pdf.

[12] F. F. Lara Martín. *Material de la asignatura Ciencias de la Computación*. Depto. Ciencias de la Computación e Inteligencia Artifical. Univ. de Sevilla. 2015.

[13] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1993.

[14] Robert W. Ritchie. "Classes of predictably computable functions". En: *Trans. Amer. Math. Soc.* 106.6 (1963), págs. 139-173. URL: http://www.ams.org/journals/tran/1963-106-01/S0002-9947-1963-0158822-2/S0002-9947-1963-0158822-2.pdf.

[15] Julia Robinson. "Definability and decision problems in arithmetic". En: *The Journal of Symbolic Logic* 14.02 (1949), págs. 98-114. URL: http://dx.doi.org/10.2307/2266510.