COMPUTABILIDAD Y COMPLEJIDAD

Guillermo Morales-Luna
Departamento de Computación
CINVESTAV-IPN
gmorales@cs.cinvestav.mx

México, D. F. a 18 de enero de 2008

Contenido

1 Conceptos básicos **1**

1.1 Pruebas por contradicción . . . 2

1.2 Inducción matemática . . . 3

1.2.1 Inducción numérica . . . 4

1.2.2 Inducción recurrente . . . 5

1.3 Enumeración de Cantor . . . 6

1.4 Lenguajes formales de programación . . . 8

1.4.1 Programas-**while** . . . 10

1.4.2 Macros . . . 10

1.4.3 Semántica de los programas-**while** . . . 12

1.5 Funciones computables . . . 13

1.6 Máquinas de Turing . . . 14

1.6.1 Introducción a las máquinas de Turing . . . 14

1.6.2 Computaciones en máquinas de Turing . . . 16

1.6.3 Modelos restringidos de máquinas de Turing . . . 16

1.6.4 Computabilidad-Turing . . . 18

1.7 Tesis de Church . . . 19

1.8 Propiedades de cerradura de funciones computables . . . 20

1.9 Numerabilidad de la clase de funciones computables . . . 22

1.9.1 Codificación de programas-**while** . . . 24

2 Funciones recursivas primitivas **29**

2.1 Conceptos básicos . . . 29

2.2 Funciones elementales . . . 34

2.2.1 Algunos otros esquemas de composición . . . 34

2.2.2 Clase de funciones elementales . . . 36

2.3 Jerarquía de Grzegorczyk . . . 39

2.3.1 Una escala de funciones . . . 39

2.3.2 Propiedades de la escala . . . 40
2.3.3 Niveles de la jerarquía . . . 43
2.4 Función de Ackermann . . . 44

3 Funciones de apareamiento y universalidad 49
3.1 Funciones de apareamiento . . . 49
3.1.1 Algunas funciones de apareamiento . . . 51
3.2 Reiteración de funciones de apareamiento . . . 53
3.3 Función β de Gödel . . . 57
3.3.1 Nociones de divisibilidad . . . 57
3.3.2 Teorema Chino del Residuo . . . 61
3.3.3 Función β de Gödel . . . 63
3.4 Universalidad . . . 65
3.4.1 Codificación de programas por números . . . 65
3.4.2 Numerabilidad de la clase de máquinas de Turing . . . 66
3.4.3 La noción de "Universalidad" . . . 67
3.4.4 Máquina Universal de Turing . . . 68
3.4.5 Universalidad en programas-**while** . . . 71

4 Teoría de la recursividad 73
4.1 Funciones recursivas . . . 73
4.2 Problema de la parada . . . 74
4.3 Decidibilidad . . . 74
4.3.1 Conjuntos decidibles . . . 74
4.3.2 Proyecciones . . . 75
4.3.3 Problema de la parada (otra vez) . . . 76
4.4 Teoremas de recursión . . . 77
4.5 Teoremas de autorreproducción . . . 78
4.6 Virus en programas-**while** . . . 78
4.6.1 Definiciones básicas . . . 79
4.6.2 Matrices infinitas . . . 79
4.6.3 Teoremas de recursión . . . 80
4.6.4 Construcción de virus . . . 81
4.6.5 Malas noticias: No hay detección universal de virus . . . 81

5 Irresolubilidad 83
5.1 Teorema de Rice . . . 83

5.2 Problema de la correspondencia de Post . . . 86
5.2.1 Presentación del Problema de Post . . . 86
5.2.2 Una aplicación de PCP . . . 89
5.3 Irresolubilidad de problemas en GLC . . . 91
5.3.1 Reducción de problemas al problema de la parada . . . 91
5.3.2 Algunos otros problemas irresolubles . . . 93
5.4 Irresolubilidad en la Aritmética . . . 94
5.4.1 Aritmética de Peano . . . 94
5.4.2 El teorema de Goodstein . . . 102
5.4.3 La conjetura de Catalan . . . 106
5.5 Clasificación de problemas irresolubles . . . 107
5.5.1 Computaciones con oráculos . . . 107
5.5.2 Enumerabilidad recursiva relativa a oráculos . . . 108

6 Teoremas de jerarquía **111**
6.1 Ordenes de funciones . . . 111
6.1.1 Definiciones básicas . . . 111
6.1.2 Algunas clases de funciones . . . 112
6.1.3 Límites inferiores . . . 113
6.2 Clases de problemas . . . 114
6.2.1 Problemas de decisión . . . 114
6.2.2 Problemas de solución . . . 114
6.2.3 Comprobadores y resolvedores . . . 114
6.3 Complejidades de tiempo y espacio . . . 115
6.3.1 Dispositivos de cómputo . . . 115
6.3.2 Tiempos y espacios . . . 115
6.4 Jerarquía en espacio . . . 117
6.5 Jerarquía en tiempo . . . 117
6.6 Algunas relaciones entre clases . . . 118
6.7 Jerarquías en clases no-deterministas . . . 119
6.8 Teorema de Borodin y consecuencias . . . 120

7 Completitud-NP **121**
7.1 Clases de problemas . . . 121
7.1.1 Reducibilidades . . . 121
7.2 Problemas difíciles y completos en clases polinomiales . . . 123
7.3 Formas proposicionales y el teorema de Cook . . . 124

7.3.1 Formas booleanas . . . 124
7.3.2 El primer problema completo-NP . . . 126
7.4 Otros problemas completos-NP en formas proposicionales . . . 128

8 Algunos problemas principales completos-NP 131
8.1 Programación lineal . . . 131
8.2 Combinatoria . . . 131
8.2.1 Atercetamientos . . . 131
8.2.2 Partición . . . 134
8.3 Problemas en gráficas . . . 136
8.4 Problemas de asignación de tareas . . . 138
8.4.1 Problemas con varios procesadores . . . 138
8.4.2 Problemas con un solo procesador . . . 139
8.5 Problemas en teoría de números . . . 142
8.5.1 Congruencias cuadráticas (CC) . . . 142
8.5.2 Ecuaciones diofantinas cuadráticas (EDC) . . . 148
8.5.3 Divisibilidad simultánea de polinomios lineales (DSP) . . . 150
8.5.4 DSP . . . 151
8.5.5 Enteros algebraicos . . . 151
8.5.6 Solubilidad de DSP . . . 152
8.5.7 Algunos otros problemas . . . 154

9 Complejidad en redes de Petri generalizadas 159
9.1 Nociones básicas . . . 159
9.2 Ejemplo . . . 161
9.3 Problemas de acotación . . . 162
9.4 Problemas de acceso . . . 163
9.4.1 Problema de acceso generalizado . . . 165

10 Algoritmos probabilísticos y su jerarquía 167
10.1 Algoritmos probabilísticos . . . 167
10.1.1 Expresiones polinomiales nulas . . . 167
10.1.2 Método de Monte-Carlo para probar primicidad . . . 168
10.2 Máquinas probabilísticas . . . 169
10.2.1 Máquinas del tipo 1 . . . 169
10.2.2 Máquinas-p . . . 170
10.2.3 Algunas inclusiones entre clases . . . 172

11 Complejidad de Kolmogorov **173**

11.1 Introducción . . . 173

11.2 Presentación de la teoría . . . 174

11.2.1 Complejidades condicionales . . . 176

11.2.2 Incomprimibilidad . . . 176

11.2.3 Descripciones auto-delimitadoras . . . 178

11.3 Estimaciones de la complejidad . . . 178

11.3.1 Kolmogorov . . . 178

11.3.2 Propiedades . . . 179

12 Trabajos a Desarrollar **181**

12.1 Primera lista de ejercicios . . . 181

12.2 Primera lista de programas . . . 183

12.3 Segunda lista de ejercicios . . . 186

12.4 Segunda lista de programas . . . 189

12.5 Tercera lista de ejercicios . . . 189

12.6 Tercera lista de programas . . . 192

Capítulo 1

Conceptos básicos

En matemáticas, la veracidad de una proposición queda asentada sólo cuando se la demuestra. Para aceptar una proposición cuantificada universalmente, es decir, de la forma "Todo x cumple $\Phi(x)$", es insuficiente demostrar que para algunos posibles "testigos" $x_1, \ldots, x_N$ se ha probado que efectivamente se cumple $\Phi(x_i)$, con $i = 1, \ldots, N$, por muy grande que sea N, pues aunque haya evidencia de que muchos puntos satisfacen al predicado Φ, ésta no basta para concluir que "todos" los puntos lo satisfacen.

Dos maneras típicas de demostrar proposiciones cuantificadas universalmente son los razonamientos por contradicción y, cuando el conjunto de puntos es numerable, por inducción. Las primeras dos secciones de este capítulo se abocan a la presentación de esos métodos de demostración.

En este capítulo introductorio presentamos posteriormente los métodos de Cantor para mostrar como numerables a los productos finitos de conjuntos numerables. Esto es de particular importancia en la Teoría de la Complejidad pues permite construir una enumeración efectiva de los "programas" en un esquema de programación que sea efectivamente "computable".

Como una primera aproximación a la noción de "computabilidad" presentamos al lenguaje de los programas-**while** como uno formal de programación, con sintaxis y semántica bien definidos, y, a través de él introducimos la noción de funciones computables.

Introducimos, luego, a las máquinas de Turing[1]. Tales máquinas aportan un enfoque alternativo al concepto de "computabilidad", sin embargo es equivalente al anterior. Probada esa equivalencia, terminaremos de presentar, de manera introductoria, a las funciones computables. Posteriormente, enunciamos la tesis de Church[2]. A las funciones computables es posible presentarlas también como elementos de clases mínimas de funciones que contienen a un cierto conjunto de funciones y que son cerradas bajo algunos esquemas de composición. De tales enfoques también nos ocuparemos en este primer capítulo.

Finalmente, veremos que la clase de programas formales, la de las máquinas de Turing y la de las funciones computables son todas numerables. Dado que la clase de todas las funciones $\mathbb{N} \to \mathbb{N}$ posee la misma cardinalidad de los números reales tenemos inmediatamente que hay muchas más funciones no-computables que computables.

[1] Alan Mathison Turing (1912-1954), matemático inglés, publicó en 1936 su célebre artículo *"On computable numbers"*, donde introdujo la noción de *máquina de Turing*.

[2] Alonzo Church (1903-(?)), lógico norteamericano, fundador de *The Journal of Symbolic Logic* en 1936. Su tesis famosa la planteó en 1956.

1.1 Pruebas por contradicción

La función $p : n \mapsto n^2 - n + 41$ define números primos al inicio.

n	$p(n)$	n	$p(n)$	n	$p(n)$	n	$p(n)$
1	41	11	151	21	461	31	971
2	43	12	173	22	503	32	1033
3	47	13	197	23	547	33	1097
4	53	14	223	24	593	34	1163
5	61	15	251	25	641	35	1231
6	71	16	281	26	691	36	1301
7	83	17	313	27	743	37	1373
8	97	18	347	28	797	38	1447
9	113	19	383	29	853	39	1523
10	131	20	421	30	911	40	1601

Sin embargo,

$$n = 41 \Rightarrow p(n) = 1681 = 41^2.$$

¡De la repetición de un patrón no necesariamente se sigue una regla general!

Es necesario presentar *pruebas* de enunciados que se quieran probar en general.

En las pruebas *por contradicción* para demostrar que una tesis ϕ se sigue de una secuencia[3] de hipótesis Φ, en símbolos $\Phi \vdash \phi$, se niega la tesis y junto con las hipótesis se deriva una contradicción. Es decir, es suficiente demostrar $\Phi \cup \{\neg\phi\} \vdash \bot$.

La aceptación de este tipo de razonamiento conlleva la aceptación del *Principio del tercero excluído* y del *Principio de consistencia de las matemáticas*, los cuales rezan:

Principio del tercero excluído. Para toda *sentencia* ϕ, es decir, para toda fórmula bien formada que sea *cerrada* (no posee variables libres), y para toda interpretación M de los símbolos de ϕ, se ha de tener que ϕ se cumple en M, en símbolos $M \models \phi$, o bien que la negación de ϕ se cumple en M, en símbolos $M \models \neg\phi$.

Principio de consistencia de las matemáticas. En toda teoría matemática T (digna de estudio), se tiene que no existe una sentencia ϕ formulada en el lenguaje de la teoría T tal que ambas ϕ y $\neg\phi$ sean demostrables en T, es decir, tal que $T \vdash \phi$ y $T \vdash \neg\phi$.

Aunque de mucho sentido común, ambos principios son cuestionables desde diversos supuestos formales. Como meras ilustraciones de estos cuestionamientos mencionamos dos de ellos:

1. El principio del tercero excluído permite demostrar de manera no constructiva la existencia de objetos.

   En efecto, veamos que existen dos números irracionales tales que uno elevado al otro da un racional.

   Sea $r = \left(\sqrt{2}\right)^{\sqrt{2}}$. Si r fuera racional, considérese $p = \sqrt{2}$ y $q = \sqrt{2}$. En otro caso, considérese $p = r$ y $q = \sqrt{2}$. En cualquier caso se tiene que ambos p y q son irracionales y p^q es racional.

   Tenemos pues demostrada así la afirmación formulada. Sin embargo, esta demostración no da ejemplos de irracionales cuya potencia de uno al otro da un racional.

2. La aritmética de Peano se realiza naturalmente en el conjunto de los números naturales. Estos conforman el modelo *estándar* de la aritmética. Sin embargo, de acuerdo con el Segundo Teorema de Incompletitud de Gödel, la misma aritmética de Peano no puede demostrar su propia consistencia.

[3] Utilizaremos el término *secuencia* como sinónimo de una *sucesión finita*.

Los siguientes teoremas se prueban por contradicción y los citamos como ejemplos de este tipo de demostraciones.

Teorema 1.1.1 (Pitágoras, 582-500 AC) $\sqrt{2}$ *no es un número racional.*

DEM. Si $p, q \in \mathbb{Z}$ fuesen dos enteros tales que $\sqrt{2} = \frac{p}{q}$, entonces

$$p^2 = 2q^2, \tag{1.1}$$

consecuentemente p^2 es un múltiplo de 2, y por esto mismo $p = 2p_1$ también lo es. En la ec. (1.1) vemos un 2 a la derecha. Falta al menos uno más, pues $p^2 = p \cdot p$ es al menos divisible por 4. Luego hemos de tener necesariamente que q^2 es un múltiplo de 2, y por esto mismo $q = 2q_1$ también lo es. Hemos pues encontrado $p_1, q_1 \in \mathbb{Z}$ tales que

$$p = 2p_1 \qquad , \qquad q = 2q_1 \qquad , \qquad \sqrt{2} = \frac{p_1}{q_1}. \tag{1.2}$$

Reiterando las relaciones (1.2) tenemos que para cualquier $n \geq 0$ existen $p_n, q_n \in \mathbb{Z}$ tales que

$$p = 2^n p_n \qquad , \qquad q = 2^n q_n \qquad , \qquad \sqrt{2} = \frac{p_n}{q_n}.$$

p y q serían pues divisibles por potencias arbitrariamente grandes de 2. Esto no es posible, dado que esos números son enteros (estándares). Por tanto, no pueden existir p y q. □

Teorema 1.1.2 (Euclides, ∼300 AC) *Hay una infinidad de números primos.*

DEM. Si ése no fuera el caso, denotemos por $P = \{p_0, p_1, \ldots, p_{n-1}\}$ al conjunto finito de, digamos que n, números primos. Consideremos el número entero $q_n = \prod_{i=0}^{n-1} p_i + 1$. Es evidente que ningún primo $p_i \in P$ puede dividir a q_n, pero, de acuerdo con el Teorema Fundamental de la Aritmética, q_n ha de ser divisible por algún número primo p_n, que acaso puede ser tal que $p_n = q_n$. El primo p_n no está en P y esto contradice que P comprendía a todos los primos. □

Teorema 1.1.3 *Si $a \in \Sigma$ y $x \in \Sigma^*$ son tales que $ax = xa$ entonces $\exists n : x = a^n$.*

DEM. Supongamos que ése no fuera el caso y que exista un símbolo distinto a a que aparezca en la palabra x. Sea k la mínima posición, contada desde la izquierda de x, donde aparezca un símbolo distinto a a, digamos b. Entonces en la palabra ax hasta antes de la posición $k+1$ sólo hay a-s, en tanto que en la palabra xa hay una b exactamente en la posición k. Por tanto no puede ocurrir que $ax = xa$. □

1.2 Inducción matemática

La *inducción matemática* provee de procedimientos de demostración de fórmulas cuantificadas universalmente, es decir, del tipo $\forall x \phi(x)$. En todo tal esquema, se verifica primero que se cumple ϕ para los casos llamados *básicos*, y después, suponiendo que se cumple para los casos anteriores, se verifica para un elemento típico x arbitrario. Este último paso es llamado "*inductivo*". Se concluye entonces que la fórmula ϕ vale para cualquier x.

Veremos aquí dos esquemas de inducción cuando el dominio de la variable x es el conjunto de los números naturales. Veremos inmediatamente después un esquema de inducción sobre conjuntos numerables definidos de manera recurrente.

1.2.1 Inducción numérica

Sea ϕ una fórmula con una variable libre x, definida en el lenguaje de la aritmética. Para demostrar la sentencia $\forall n(\phi(n))$, un primer esquema de inducción verifica primero que se cumple $\phi(0)$ y luego, suponiendo que se cumple $\phi(n)$ demuestra que se cumple $\phi(n+1)$; un segundo esquema de inducción verifica primero que se cumple $\phi(0)$ y luego, suponiendo que se cumple $\phi(m)$, para cualquier $m < n$, demuestra que se cumple $\phi(n)$. Estos esquemas, puestos como reglas de deducción quedan como sigue:

$$\textit{Esquema I} \quad : \quad \frac{\begin{array}{l}\phi(0)\\ \forall n(\phi(n) \Rightarrow \phi(n+1))\end{array}}{\forall n(\phi(n))}$$

$$\textit{Esquema II} \quad : \quad \frac{\begin{array}{l}\phi(0)\\ \forall n(\forall m < n(\phi(m)) \Rightarrow \phi(n))\end{array}}{\forall n(\phi(n))}$$

Teorema 1.2.1 *Ambos esquemas son equivalentes.*

Dem.

1. *Esquema I* $\Rightarrow$ *Esquema II*): Supongamos válido el *Esquema I.* Sea ϕ una fórmula tal que se cumplen

$$\phi(0) \qquad \& \qquad \forall n(\forall m < n(\phi(m)) \Rightarrow \phi(n)). \tag{1.3}$$

   Mostremos primero que

$$\forall n(\phi(0) \Rightarrow \forall m \leq n : \phi(m)). \tag{1.4}$$

   Para esto usemos el *Esquema I* aplicado a la fórmula $\phi_1(n) \equiv [\phi(0) \Rightarrow \forall m \leq n : \phi(m)]$.

   $\phi_1(0)$ se cumple en virtud de que para cualquier fórmula ψ, la fórmula $(\psi \Rightarrow \psi)$ es un teorema del cálculo de predicados puro.

   Ahora, supongamos que vale $\phi_1(n)$. Hemos de probar que vale $\phi_1(n+1)$.

   Supongamos $\phi(0)$. Por $\phi_1(n)$ se tiene que es válido $(\forall m \leq n : \phi(m))$. Por (1.3) se cumple también $\phi(n+1)$. Así pues, la fórmula $(\forall m \leq (n+1) : \phi(m))$ es válida, con lo cual se ve que, en efecto, la fórmula $\phi_1(n+1)$ se cumple.

   Demostremos ahora que vale $\forall n\,[\phi(n) \Rightarrow \phi(n+1)]$. Dado un n cualquiera, suponiendo que se cumple $\phi(n)$, puesto que $\phi(0)$ es cierto, se cumple por (1.4) que $\forall m \leq n : \phi(m)$ y por (1.3) resulta ya demostrado $\phi(n+1)$.

   Por el *Esquema I*, resulta como una consecuencia $\forall n \phi(n)$.

   Tenemos pues que se cumple el *Esquema II.*

2. *Esquema II* $\Rightarrow$ *Esquema I*): Supongamos válido el *Esquema II.* Sea ϕ una fórmula tal que se cumplen

$$\phi(0) \qquad \& \qquad \forall n(\phi(n) \Rightarrow \phi(n+1)). \tag{1.5}$$

   Resulta evidente que se cumple la fórmula $\forall n(\forall m < n(\phi(m)) \Rightarrow \phi(n))$, pues, para cada $n > 0$ la condición $[\forall m < n : \phi(m)]$ subsume a la fórmula $\phi(n-1)$, la cual, de acuerdo con la segunda fórmula en (1.5) de este apartado implica $\phi(n)$.

   Por el *Esquema II*, resulta como una consecuencia $\forall n \phi(n)$.

   Tenemos pues que se cumple también el *Esquema I.* □

1.2.2 Inducción recurrente

Este esquema se utiliza para demostrar predicados cuantificados universalmente definidos sobre conjuntos determinados constructivamente de manera recurrente.

Sea T un conjunto numerable.

Sea $A_0 \subset T$ un conjunto inicial. Supongamos $A \subset T$ definido recurrentemente con las reglas siguientes:

$$\begin{aligned} \textit{Regla}_0 \quad &: \qquad \forall a : a \in A_0 \quad \Rightarrow \quad a \in A \\ \textit{Regla}_j \quad &: \quad \forall a_1, \ldots, a_{j_k} : a_1, \ldots, a_{j_k} \in A \quad \Rightarrow \quad \Phi_j(a_1, \ldots, a_{j_k}) \in A \end{aligned}$$

para $j = 1, \ldots, m$, donde cada función $\Phi_j : T^{j_k} \to T$ es, en la práctica, una *regla de composición.*

Para todo predicado Ψ se tiene el esquema de demostración siguiente:

$$\textit{Esquema III} \quad : \quad \frac{\begin{array}{lll} \forall a \in A_0 & : & \Psi(a) \\ \forall a_1, \ldots, a_{j_k} & : & (\forall i : \Psi(a_i)) \Rightarrow \Psi(\Phi_j(a_1, \ldots, a_{j_k})) \end{array}}{\forall a \in A \quad : \quad \Psi(a)}$$

Como un ejemplo de construcción mediante este tipo de reglas, consideremos al siguiente:

Ejemplo. Sea T el conjunto formado por las palabras (de longitud finita) sobre el alfabeto $(0 + 1)$, i.e. $T = (0 + 1)^*$. Definimos el conjunto de palabras 0-*preponderadas* de manera recurrente como sigue:

$$\begin{aligned} \textit{Regla}_0 \quad &: \qquad\qquad\qquad\qquad\qquad\quad \textit{nil} \text{ es 0-preponderada,} \\ \textit{Regla}_1 \quad &: \qquad \alpha \text{ es 0-preponderada} \quad \Rightarrow \quad 0\alpha \text{ es 0-preponderada,} \\ \textit{Regla}_2 \quad &: \quad \alpha_1, \alpha_2 \text{ son 0-preponderadas} \quad \Rightarrow \quad 0\alpha_1 1\alpha_2 \text{ es 0-preponderada.} \end{aligned}$$

El conjunto $A \subset T$ de palabras 0-preponderadas se ajusta a la construcción recurrente, mencionada al inicio de esta sección, donde $A_0 = \{nil\}$ consta únicamente de la palabra vacía y hay dos reglas de composición Φ_1, Φ_2 dadas por sendas reglas. Más adelante veremos que una palabra es 0-preponderada si cualquier prefijo de ella posee más ceros que unos.

Proposición 1.2.1 *El esquema III se prueba mediante el esquema II por inducción en el número de reglas para generar un elemento en A.*

Dem. Con la notación utilizada en la formulación del *Esquema III*, sea

$$\phi(n) \equiv [\Psi(a) \text{ se cumple siempre que } a \in T \text{ se obtenga con a lo sumo } (n + 1) \text{ aplicaciones de las reglas que definen a } A]$$

Como A se define únicamente por las reglas enlistadas, la fórmula $\forall a \in A : \Psi(a)$ es lógicamente equivalente a la fórmula $\forall n : \phi(n)$. Mostremos pues que esta última es válida a partir de las hipótesis

$$\forall a \in A_0 : \Psi(a) \qquad , \qquad \forall a_1, \ldots, a_{j_k} : (\forall i : \Psi(a_i)) \Rightarrow \Psi(\Phi_j(a_1, \ldots, a_{j_k}))$$

La primera de estas ecuaciones equivale a $\phi(0)$. La segunda es claramente equivalente a

$$\forall n(\forall m < n(\phi(m)) \Rightarrow \phi(n)).$$

Por el *Esquema II* tenemos que se cumple $\forall n : \phi(n)$, y con esto queda demostrado el *Esquema III.* □

Ejemplo. Retomando nuestro ejemplo anterior, sea A el conjunto de palabras 0-preponderadas. Sea $\Psi(\alpha)$ la proposición

$$\Psi(\alpha) \equiv [\text{En cualquier prefijo de } \alpha \text{ el número de 1's a lo sumo coincide con el de 0's.}]$$

	0	1	2	3	4	5	6	7	8	9	10	
0	0	1	3	6	10	15	21	28	36	45	55	...
1	2	4	7	11	16	22	29	37	46	56	67	...
2	5	8	12	17	23	30	38	47	57	68	80	...
3	9	13	18	24	31	39	48	58	69	81	94	...
4	14	19	25	32	40	49	59	70	82	95	109	...
5	20	26	33	41	50	60	71	83	96	110	125	...
6	27	34	42	51	61	72	84	97	111	126	142	...
7	35	43	52	62	73	85	98	112	127	143	160	...
8	44	53	63	74	86	99	113	128	144	161	179	...
9	54	64	75	87	100	114	129	145	162	180	199	...
10	65	76	88	101	115	130	146	163	181	200	220	...
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

Tabla 1.1: Función de *enumeración de Cantor*.

Procedamos de acuerdo con el *Esquema III* para ver que $\forall\alpha \in A : \Psi(\alpha)$.

Inicialmente tenemos, trivialmente, que para la palabra vacía se cumple Ψ, es decir, vale $\Psi(\text{nil})$.

Ahora, de manera inductiva:

Regla$_1$: Supongamos que la palabra 0-preponderada 0α se obtiene de la aplicación de la regla 1 a la palabra 0-preponderada α, obtenida ésta con un menor número de aplicaciones de las reglas de A. Sea α_1 cualquier prefijo (no-vacío) de 0α. Entonces $\alpha_1 = 0\alpha_1'$, donde α_1' es un prefijo de α. Sean k_0 y k_1 los números de 0's y 1's en α_1'. Obviamente, $k_0 + 1$ y k_1 son los números de 0's y 1's en α_1. Por la hipótesis de inducción tenemos que $k_1 \leq k_0$. Cuantimás $k_1 \leq k_0 + 1$. Así pues se cumple $\Psi(0\alpha)$.

Regla$_2$: Supongamos que la palabra 0-preponderada $0\alpha_1 1\alpha_2$ se obtiene de la aplicación de la regla 2 a las palabras 0-preponderadas α_1, α_2, obtenida cada una de éstas con un menor número de aplicaciones de las reglas de A. Con una argumentación similar a la anterior se muestra que vale $\Psi(0\alpha_1 1\alpha_2)$.

1.3 Enumeración de Cantor

A finales del siglo XIX y principios del XX el matemático alemán Georg Cantor (1845-1918) estableció las bases de los números transfinitos. El primer numeral infinito, denotado por ω_0, puede ser identificado con la cardinalidad de los números naturales. En el estudio de las cardinalidades es importante reconocer como *equipotentes*, es decir "con la misma cardinalidad", a cualesquiera dos conjuntos que, en efecto, lo sean. Una gran contribución de Cantor es su análisis de los conjuntos *numerables*.

Consideremos la función $c : \mathbb{N}^2 \to \mathbb{N}$, $c : (x, y) \mapsto \frac{1}{2}(x + y)(x + y + 1) + x$. En la tabla (1.1) presentamos como una matriz (infinita) los valores de esta función. En la entrada correspondiente al renglón y y a la columna x presentamos el valor $c(x, y)$. Obsérvese que los valores de c son consecutivos a lo largo de cada diagonal $x + y = d$, $d \in \mathbb{N}$. Obsérvese también que c está dado por un polinomio cuadrático.

De los valores mostrados en la tabla (1.1) se puede tener una idea muy precisa de la enumeración que hace c de $\mathbb{N}^2$. Resulta evidente que cada entrada de la tabla tiene asociado un número natural y, viceversa, cada número natural estará en correspondencia con alguna entrada en la tabla. c es pues una *biyección*, es decir, es una función inyectiva y suprayectiva. c se dice ser la *enumeración de Cantor*.

Así pues, $\mathbb{N}^2$ es *numerable*, es decir, posee la misma cardinalidad que $\mathbb{N}$. A partir de esto es fácil ver por inducción que $\forall n > 0, \mathbb{N}^n$ es también numerable. De hecho, la enumeración de Cantor da procedimientos para enumerar a cada potencia $\mathbb{N}^n$.

Secuencia	Código
$[1,2]$	37
$[1,2,3]$	2082
$[1,2,3,4]$	2591229
$[1,2,3,4,5]$	3374953984715
$[1,2,3,4,5,6]$	5695183504482489143926610
$[1,2,3,4,5,6,7]$	1621755757492238630142051988670435611794067575773 2
$[1,2,3,4,5,6,7,8]$	1315045868479612356871818745780631171143294098975\ 665358313669490036179642240153575195006840456554 68
$[1,2,3,4,5,6,7,8,9]$	864672818102648960261040653715831867092837278 6\ 737024641130379069394221138489756289944296330853 10\ 804523264790074666659732320017335210243287151463 48\ 276749651527449688793203577465575382646833155807 38
$[1,2,3,4,5,6,7,8,9,10]$	373829541182788327738952442182366863389086\ 465504374005023581180645307530245500626409060671 70\ 928616739335301863382037085201145976272239453658 98\ 126284612819050821939596644888791720918094343075 68\ 163177203802091671190093130300866436550621259428 27\ 415066831038169370702234679647241685978053635205 14\ 192657137294322802013713613628921762213636517748 68\ 477032110299949407246036473543852971522319704739 85

Tabla 1.2: Valores de $f^*(\{1,\ldots,i\})$ para $i = 1,\ldots,10$.

Código:	Sucesión codificada
1953 :	(62)
1954 :	$(6,4)$
1955 :	$(2,0,5)$
1956 :	$(0,1,1,6)$
1957 :	$(0,0,0,2,7)$
1958 :	$(0,0,0,1,0,8)$
1959 :	$(0,0,0,0,0,1,9)$
1960 :	$(0,0,0,0,0,0,0,10)$

Tabla 1.3: Valores de $(f^*)^{-1}(n)$ para $n = 1953,\ldots,1960$.

En efecto, reiterando la enumeración de Cantor, hagamos

$$\begin{array}{rcl} & & f_1 : x \mapsto x \\ \forall n \geq 0 & : & f_{n+1} : (\mathbf{x}, x) \mapsto f_{n+1}(\mathbf{x}, x) = c(f_n(\mathbf{x}), x) \\ & & f^* : \mathbf{x} \mapsto f^*(\mathbf{x}) = c(\mathrm{Long}(\mathbf{x}) - 1, f_{\mathrm{Long}(\mathbf{x})}(\mathbf{x})) \end{array}$$

Es fácil ver también que la función $f^* : \mathbb{N}^* = \bigcup_{n\geq 0} \mathbb{N}^n \to \mathbb{N}$ es una biyección.

Para cada secuencia de números naturales $\mathbf{x} \in \mathbb{N}^*$ el número $\lceil \mathbf{x} \rceil = f^*(\mathbf{x})$ se dice ser el *código* de la secuencia $\mathbf{x}$. Recíprocamente, dado un número $n \in \mathbb{N}$ la secuencia $\mathbf{x} \in \mathbb{N}^*$ tal que $\lceil \mathbf{x} \rceil = n$ es la *secuencia codificada por* n y, en tal caso, escribimos $\mathbf{x} = \lfloor n \rfloor$.

En la tabla (1.2) se ejemplifica la relación entre secuencias y códigos. En la tabla (1.3) se ejemplifica la relación inversa entre códigos y secuencias.

Resumiendo, tenemos la

Input: n: Length of $\mathbf{x}$.
$\mathbf{x}$: Sequence to be codified.
Output: $r = \lceil \mathbf{x} \rceil$

```
Algorithm DirCode
{  case n of
   1:     r := x ;
   2:     r := c(x) ;
   else : {  i := 2 ;
             r := c(x_1, x_2) ;
             while i < n do { i++ ; r := c(r, x_i) }
          }
}
```

Figura 1.1: Cálculo de la función f_n.

Input: $\mathbf{x}$: Sequence to be codified.
Output: $r = \lceil \mathbf{x} \rceil$

```
Algorithm DirCodeStar
{  n := Length(x) ; r_1 := DirCode(n, x) ; r := c(n − 1, r_1)
}
```

Figura 1.2: Cálculo de la función f^*.

Proposición 1.3.1 *Para cada $n > 0$ la potencia cartesiana $\mathbb{N}^n$ es numerable, y lo es también el conjunto $\mathbb{N}^*$ consistente de las secuencias de números naturales. Más aún cada uno de estos conjuntos posee una enumeración efectivamente calculable con inversa también efectivamente calculable.*

$\mathbb{N}^2$ es enumerable mediante la enumeración c que está dada por un polinomio y es efectivamente calculable.

En la figura (1.1) mostramos un seudocódigo, que se explica por sí mismo, para calcular a la función f_n. En la figura (1.2) mostramos un seudocódigo para calcular a la función f^*. En la figura (1.3) mostramos un seudocódigo para calcular a la inversa de la enumeración c. En la figura (1.4) mostramos un seudocódigo recursivo en n para calcular a la inversa de cada una de las funciones f_n. Finalmente, en la figura (1.5) mostramos un seudocódigo para calcular a la inversa de la función f^*.

1.4 Lenguajes formales de programación

Todo lenguaje de programación consta de una sintaxis bien determinada que permite identificar a los programas *bien formados* y de una semántica que permite interpretar procedimentalmente a los programas. Toda sintaxis se establece sobre un *alfabeto*, es decir, sobre un conjunto finito de símbolos, mediante *reglas gramaticales* o *producciones* para generar diversas clases de palabras sobre el alfabeto.

Input: r: The code of a pair.
Output: $\mathbf{x} = c^{-1}(r)$

```
Algorithm InvCantor
{  Let d be such that (1/2)d(d+1) ≤ r < (1/2)(d+1)(d+2) ;
   x := r − (1/2)d(d+1) ;
   y := d − x ;
   x := (x, y)
}
```

Figura 1.3: Cálculo de la inversa de c.

Input: r: The code of a sequence.
n: The length of the sequence to be decodified.
Output: $\mathbf{x} = \lfloor r \rfloor$

```
Algorithm InvCode
{  if n = 1 then x := r
   else
   {  (a, b) := InvCantor(r) ;
      x_1 := InvCode(n − 1, a) ;
      x := Append(x_1, b)
   }
}
```

Figura 1.4: Cálculo de la inversa de f_n.

Input: r: The code of a sequence.
Output: $\mathbf{x} = \lfloor r \rfloor$

```
Algorithm InvCodeStar
{  (n_1, r_1) := InvCantor(r) ; n := n_1 + 1 ; x := InvCode(n_1, r_1)
}
```

Figura 1.5: Cálculo de la inversa de f^*.

$$
\begin{array}{rcl}
\langle \mathit{Prueba} \rangle & ::= & \langle \mathit{Variable} \rangle \neq 0 | \langle \mathit{Variable} \rangle = 0 \\
\langle \mathit{Instrucción} \rangle & ::= & \mathbf{nil} | \langle \mathit{Variable} \rangle + + | \langle \mathit{Variable} \rangle - - | \\
 & & \mathbf{while}\ \langle \mathit{Prueba} \rangle\ \mathbf{do}\ \langle \mathit{Programa} \rangle \\
\langle \mathit{Rutina} \rangle & ::= & \langle \mathit{Instrucción} \rangle\ ;\ \langle \mathit{Instrucción} \rangle | \\
 & & \langle \mathit{Instrucción} \rangle\ ;\ \langle \mathit{Rutina} \rangle \\
\langle \mathit{Programa} \rangle & ::= & \langle \mathit{Instrucción} \rangle | \{ \langle \mathit{Rutina} \rangle \}
\end{array}
$$

Figura 1.6: Reglas de producción en el lenguaje de los programas-**while**.

1.4.1 Programas-while

Consideremos los siguientes conjuntos de símbolos:

$$
\begin{array}{rcl}
\mathit{Alfabeto} & = & \{0\} \cup \mathit{Variables} \cup \mathit{Especiales} \\
\mathit{Especiales} & : & \}, \{, ++, --, =, \neq, ;, \mathbf{while}, \mathbf{do}
\end{array}
$$

Los símbolos de funciones unarias $++$ y $--$ se interpretan naturalmente:

$$
x++ = x+1 \qquad x-- = \begin{cases} x-1 & \text{si } x > 0 \\ 0 & \text{en otro caso.} \end{cases}
$$

La colección de programas-**while** se define recurrentemente. En la figura (1.6) resumimos la sintaxis de los programas-**while** y de ahí se ve que los programas-**while** provienen de una sintaxis muy sencilla.

1.4.2 Macros

A pesar de su sencillez los programas-**while** son muy expresivos. Definimos en esta sección algunos procedimientos mediante correspondientes programas-**while**. Ulteriormente, en la presentación de estos programas nos referiremos a algunos de los *macros* aquí definidos. Sin embargo, el lector ha de tener en cuenta que, estrictamente, toda vez que aparezca un macro se ha de sustituir ese macro por el esquema que aquí se presenta para tener un programa-**while**.

Asignaciones

Asignar valores a variables.

1. $x := 0$:

   $$\boxed{\mathbf{while}\ x \neq 0\ \mathbf{do}\ x--}$$

2. $x := y$:

   $$\boxed{\begin{array}{l} \{\quad x := 0\ ; \\ \quad\ \mathbf{while}\ y \neq 0\ \mathbf{do}\ \{\ x++\ ;\ y--\ \} \\ \} \end{array}}$$

3. $x := n$, donde n es un número natural.
   En efecto, hagamos $\boxed{r_0 \equiv\ \mathbf{while}\ x \neq 0\ \mathbf{do}\ \ x--}$ y para cada $n \in \mathbb{N}$, sea $\boxed{r_{n+1} \equiv\ r_n; x++}$. Entonces el macro $x := n$ coincide con el programa $\boxed{\{r_n\}}$.

Operaciones aritméticas

Algunos macros para calcular funciones elementales son los siguientes.

4. Suma: $z := x + y$

$$\boxed{\begin{array}{l}\{\quad z := x\ ;\\ \quad \textbf{while}\ y \neq 0\ \textbf{do}\ \{\ z{+}{+}\ ;\ y{-}{-}\ \}\\ \}\end{array}}$$

5. Diferencia acotada: $z := x \dot{-} y = \begin{cases} x - y & \text{si } x \geq y \\ 0 & \text{en otro caso.} \end{cases}$

$$\boxed{\begin{array}{l}\{\quad z := x\ ;\\ \quad \textbf{while}\ y \neq 0\ \textbf{do}\ \{\ z{-}{-}\ ;\ y{-}{-}\ \}\\ \}\end{array}}$$

6. Producto: $z := x * y$

$$\boxed{\begin{array}{l}\{\quad z := 0\ ;\\ \quad \textbf{while}\ y \neq 0\ \textbf{do}\ \{\ z = z + x\ ;\ y{-}{-}\ \}\\ \}\end{array}}$$

7. División entera: $z := x$ `div` y

$$\boxed{\begin{array}{l}\{\quad z := 0\ ;\\ \quad w := x \dot{-} y\ ;\\ \quad \textbf{while}\ w \neq 0\ \textbf{do}\ \{\ z{+}{+}\ ;\ w := w \dot{-} y\ \}\\ \}\end{array}}$$

8. Parte entera del logaritmo en base 2: $z := \lfloor \log_2 x \rfloor$

$$\boxed{\begin{array}{l}\{\quad z := 0\ ;\\ \quad w := 1\ ;\\ \quad v := x \dot{-} w\ ;\\ \quad \textbf{while}\ v \neq 0\ \textbf{do}\ \{\ z{+}{+}\ ;\ w := 2 * w\ ;\ v := x \dot{-} w\ \}\\ \}\end{array}}$$

Pruebas compuestas

Las pruebas compuestas se definen como sigue

$$\begin{array}{rcl}\langle \mathit{TérmBas} \rangle & ::= & \langle \mathit{Variable} \rangle | \langle \mathit{Número} \rangle \\ \langle \mathit{PruebaComp} \rangle & ::= & \langle \mathit{TérmBas} \rangle = \langle \mathit{TérmBas} \rangle | \\ & & \langle \mathit{TérmBas} \rangle \neq \langle \mathit{TérmBas} \rangle | \\ & & \langle \mathit{TérmBas} \rangle < \langle \mathit{TérmBas} \rangle | \\ & & \langle \mathit{PruebaComp} \rangle \wedge \langle \mathit{PruebaComp} \rangle | \\ & & \langle \mathit{PruebaComp} \rangle \vee \langle \mathit{PruebaComp} \rangle | \\ & & \neg \langle \mathit{PruebaComp} \rangle \end{array}$$

Proposición 1.4.1 *La proposición*

$$\boxed{\textbf{while } \langle \mathit{PruebaComp} \rangle \textbf{ do } \langle \mathit{Programa} \rangle}$$

*es un macro en el lenguaje de los programas-***while***.*

DEM. Para cada prueba P construimos una expresión E_P tal que

$$\begin{array}{lcl} P(\mathbf{X}) = \mathit{Verdadero} & \Leftrightarrow & E_P(\mathbf{X}) = 1 \\ P(\mathbf{X}) = \mathit{Falso} & \Leftrightarrow & E_P(\mathbf{X}) = 0 \end{array}$$

De manera recursiva:

$$\begin{array}{lcl} P(X,Y) = (X < Y) & \Rightarrow & E_P(X,Y) = (Y \dot{-} X) \dot{-} (Y \dot{-} X) - - \\ P(\mathbf{X}) = P_1(\mathbf{X}) \wedge P_2(\mathbf{X}) & \Rightarrow & E_P(\mathbf{X}) = (E_{P_1}(\mathbf{X}) + E_{P_2}(\mathbf{X})) - - \\ P(\mathbf{X}) = P_1(\mathbf{X}) \vee P_2(\mathbf{X}) & \Rightarrow & E_P(\mathbf{X}) = (E_{P_1}(\mathbf{X}) + E_{P_2}(\mathbf{X})) \dot{-} (E_{P_1}(\mathbf{X}) + E_{P_2}(\mathbf{X})) - - \\ P(\mathbf{X}) = \neg P_1(\mathbf{X}) & \Rightarrow & E_P(\mathbf{X}) = 1 \dot{-} E_{P_1}(\mathbf{X}) \\ P(X,Y) = (X = Y) & \Rightarrow & E_P(X,Y) = E_{\neg(X<Y)\wedge\neg(Y<X)}(X,Y) \\ P(X,Y) = (X \neq Y) & \Rightarrow & E_P(X,Y) = E_{\neg(X=Y)}(X,Y) \end{array}$$

□

Macros de programación

Proposición 1.4.2 *Las proposiciones (***if - then** *), (***if - then - else** *), (***repeat -until** *) son macros en los programas-***while***.*

DEM. Como un mero ejemplo, tenemos que al esquema [**if** PC **then** *Prog*] se le puede expresar como sigue

$$\boxed{\begin{array}{l} \{ \quad v := 0 \ ; \\ \quad \textbf{while } PC \wedge (v = 0) \textbf{ do } \{ \ \mathit{Prog} \ ; \ v{+}{+} \ \} \\ \} \end{array}}$$

□

Proposición 1.4.3 *La proposición (***for** $\langle \mathit{Ctr} \rangle = c_{\mathit{inicial}}$ **to** c_{final} **do** *Prog) es un macro en los programas-***while***.*

DEM. De manera más bien tosca, tenemos que el esquema [**for** $w = c_{\mathit{inicial}}$ **to** c_{final} **do** $\mathit{Prog}(w)$] es equivalente al siguiente

$$\boxed{\begin{array}{l} \{ \quad v := c_{\mathit{final}} \dot{-} c_{\mathit{inicial}} \ ; \ w := c_{\mathit{inicial}} \ ; \\ \quad \textbf{while } v \neq 0 \textbf{ do } \{ \ \mathit{Prog}(w) \ ; \ w{+}{+} \ ; \ v{-}{-} \ \} \\ \} \end{array}}$$

1.4.3 Semántica de los programas-while

En cuanto a la semántica de los programas-**while**, introduciremos primero la noción de estados y a cada programa lo veremos como un proceso que transforma estados en estados.

Si P es un programa-**while** y $[x_1 \dots x_k]$ es la lista de variables que aparecen en P, entonces $\mathbb{N}^k$ es el *espacio de estados* o *configuraciones* de P. Los estados se transforman entre sí mediante la aplicación de programas o instrucciones.

Si P es un programa o instrucción y $\mathbf{x} \in \mathbb{N}^k$ es un estado, denotaremos por $\mathbf{y} = P(\mathbf{x})$ al estado al cual P transforma la entrada $\mathbf{x}$.

Introducimos como un estado suplementario al *estado de indefinición* o *indeterminado*, denotado usualmente por el símbolo $\perp$.

Reglas de transformación

$$
\begin{array}{lcl}
P \equiv [x_i++] & \Rightarrow & \forall j : y_j = \left\{ \begin{array}{ll} x_i++ & \text{si } j = i \\ x_j & \text{si } j \neq i \end{array} \right. \\
P \equiv [x_i--] & \Rightarrow & \forall j : y_j = \left\{ \begin{array}{ll} x_i-- & \text{si } j = i \\ x_j & \text{si } j \neq i \end{array} \right. \\
P \equiv [P_1; P_2] & \Rightarrow & \mathbf{y} = P_2(P_1(\mathbf{x})) \\
P \equiv [\textbf{while } \phi(\mathbf{x}) \textbf{ do } P_1] & \Rightarrow & \mathbf{y} = \left\{ \begin{array}{ll} P_1^n(\mathbf{x}) & \text{si } \exists n = \text{Min}\{m | \neg\phi(P_1^m(\mathbf{x}))\} \\ \perp & \text{en otro caso.} \end{array} \right.
\end{array}
$$

Escribiremos

$$
\begin{array}{rcl}
P(\mathbf{x}) \uparrow & \Leftrightarrow & P(\mathbf{x}) = \perp \\
P(\mathbf{x}) \downarrow & \Leftrightarrow & P(\mathbf{x}) \neq \perp
\end{array}
$$

En el primer caso, diremos que el programa P *diverge* para los datos $\mathbf{x}$. En el segundo, naturalmente, diremos que el programa P *converge*.

1.5 Funciones computables

Las funciones computables son las que se calculan por programas-**while**.

Si P es un programa-**while** y $[x_1 \dots x_k]$ es la lista de variables en P entonces consideraremos a las primeras variables como variables de *entrada* y a las últimas como de *salida*, de manera sólo un poco más precisa, para $n, m \in N$ consideraremos

$$
\begin{array}{rcl}
\mathbf{x}_n = [x_1 \dots x_n] & : & \text{lista de variables de } \textit{entrada} \\
\mathbf{x}^{-m} = [x_{k-m+1} \dots x_{k-1} x_k] & : & \text{lista de variables de } \textit{salida}
\end{array}
$$

Diremos que el programa P *calcula*, o *computa*, a la función

$$
\begin{array}{rcl}
f_{P,n,m} : \mathbb{N}^n & \rightarrow & \mathbb{N}^m \\
\mathbf{x}|_n & \mapsto & P\left(\mathbf{x}|_n\right)^{-m}
\end{array}
$$

donde $\mathbf{x}|_n = \left\{ \begin{array}{ll} [\mathbf{x}, \mathbf{0}_{k-n}] & \text{si } n \leq k \\ \pi_{[1\dots k]}(\mathbf{x}) & \text{en otro caso.} \end{array} \right.$

En este caso definimos,

$$
\begin{array}{lcl}
\textit{Dominio de } f_{P,n,m} & \text{dom}_n(f_{P,n,m}) & = \{\mathbf{x} \in \mathbb{N}^n | P\left(\mathbf{x}|_n\right) \downarrow\} \\
\textit{Imagen de } f_{P,n,m} & \text{ima}_m(f_{P,n,m}) & = \{\mathbf{y} \in \mathbb{N}^m | \exists \mathbf{x} \in \mathbb{N}^n : \mathbf{y} = P\left(\mathbf{x}|_n\right)^{-m}\}
\end{array}
$$

En adelante omitiremos los subíndices n, m y k.

$$f_P \ total \Leftrightarrow \mathrm{dom}_n(f_P) = \mathbb{N}^n.$$

Una función $f : \mathbb{N}^n \to \mathbb{N}^m$ se dice ser *computable* si existe un programa-**while** tal que $f = f_P$.

1.6 Máquinas de Turing

Las *máquinas de Turing* son autómatas que pueden ser vistos como mecanismos formales de cómputo y, en términos de ellas, plantear de manera alternativa la noción de *computabilidad*.

1.6.1 Introducción a las máquinas de Turing

Estas son autómatas finitas con dos pilas que tienen un *tope* común. Ambas pilas forman una *cinta*. El tope común es la *casilla escudriñada (scanned cell)*, una pila es la parte de la cinta a la derecha de la casilla escudriñada y la otra pila es su parte izquierda.

De manera más extensa, podemos plantear a las máquinas de Turing como sigue:

Una máquina de Turing M consta

- de una cinta lineal no-acotada conformada por *casillas*, de hecho la cinta puede ponerse en correspondencia con el conjunto $\mathbb{Z}$ de los números enteros. La única restricción sobre la cinta es que, en todo momento, todas sus casillas están en "blanco" salvo un número finito de casillas (aunque ese número finito puede ser arbitrariamente grande),
- de un *mecanismo de control* que es propiamente un autómata que puede asumir diversos *estados*,
- de una cabeza *lectora* que le permite escudriñar, en cada instante, una casilla en la cinta, y
- de un *programa*, el cual es una lista de *quintuplas*.

Cada quintupla de M es de la forma $qs \to ptm$ donde $q, p \in$ Estados , $s, t \in$ Alfabeto , $m \in \{\mathrm{Izqu}, \mathrm{Dere}, \mathrm{Alto}\}$. La connotación que se le asigna a esa quintupla es la siguiente:

> "Si en el estado q se lee en la cinta el símbolo s, entonces primero se sustituye a s por t, luego se pasa al estado p y, al final, se pasa a examinar la casilla adyacente en la dirección de m".

Como una primera representación de una máquina M, podemos identificarla con su propio programa. De hecho, al ordenar al conjunto EA = Estados $\times$ Alfabeto, la lista de quintuplas $M = \{qs \to ptm\}_{qs \in \mathrm{EA}}$ puede reescribirse como la lista de "tercetas" $M = \{ptm\}_{qs \in \mathrm{EA}}$, la cual será propiamente el "programa" de la máquina de Turing.

Ejemplo: Cadenas equilibradas de paréntesis

Para tener una cadena equilibrada, cada ")" en ella ha de "empatar" con un correspondiente "(" que lo anteceda. Para reconocer cadenas equilibradas, de manera general, a cada "(" ya empatado se le marcará por una A como ya revisado y a cada ")" se le marcará por una B. Procederemos pues como sigue:

1. Inicialmente, se busca el primer ")", yendo de izquierda a derecha.
2. Por cada ")",

$1, x; 1, x, Der$	:	con cualquier cosa, salvo ")", avánce se a la derecha, $x \in \{(, b, A, B\}$,
$1,); 2, B, Izq$	:	con el primer ")", márquese y retrocédase,
$1, b; 3, b, Izq$	:	si la lista se acaba, revísese que todo haya sido marcado,
$2, y; 2, y, Izq$	:	con cualquier cosa, salvo "(", continúese el retroceso, $y \in \{), b, A, B\}$,
$2, (; 1, A, Der$	:	márquese el "(" que empata y repítase el ciclo,
$2, b; 4, No, Alt$	:	se termina la cadena y no existe el correspondiente "(". No hay equilibrio.
$3, x; 3, x, Izq$	:	retrocédase ignorando las marcas, $x \in \{A, B\}$,
$3, (; 3, No, Alt$	:	si quedare un "(", éste ya no podría empatarse. No hay equilibrio.
$3, b; 3, Sí, Alt$	:	se agotó la cadena y todo se marcó. Sí hay equilibrio.

Tabla 1.4: Máquina de Turing para reconocer cadenas equilibradas de paréntesis.

$()(())$	$AB(A\mathbf{1}B)$	$ABAAB\mathbf{3}B$
$\mathbf{1}()(())$	$AB(AB\mathbf{1})$	$ABAA\mathbf{3}BB$
$(\mathbf{1})(())$	$AB(A\mathbf{2}BB$	$ABA\mathbf{3}ABB$
$\mathbf{2}(B(())$	$AB(\mathbf{2}ABB$	$AB\mathbf{3}AABB$
$A\mathbf{1}B(())$	$AB\mathbf{2}(ABB$	$A\mathbf{3}BAABB$
$AB\mathbf{1}(())$	$ABA\mathbf{1}ABB$	$\mathbf{3}ABAABB$
$AB(\mathbf{1}())$	$ABAA\mathbf{1}BB$	$\mathbf{3}bABAABB$
$AB((\mathbf{1}))$	$ABAAB\mathbf{1}B$	[Sí]$ABAABB$
$AB(\mathbf{2}(B)$	$ABAABB\mathbf{1}b$	

Tabla 1.5: Ejemplo de cómputo en la máquina de Turing que reconoce cadenas equilibradas de parétesis.

(a) se marca con B,

(b) se busca hacia la izquierda el primer "(" que lo empate,

(c) si no se encontrare tal "(" la cadena está desequilibrada. En otro caso, el "(" se marca con A y se repite este ciclo.

3. Si quedaren "(" no empatados, la cadena está desequilibrada. En otro caso se tiene equilibrio y se termina el proceso.

El procedimiento queda descrito como una máquina de Turing por la lista de quíntuplas mostrada en la tabla 1.4. Los cuatro estados de la máquina tienen una interpretación evidente:

1 : avanza a la derecha buscando ")",
2 : retrocede a la izquierda buscando "(",
3 : revisa que todo haya sido marcado,
4 : estado terminal.

Como un mero ejemplo, en la tabla 1.5 se ve la computación correspondiente a una cierta cadena dada. En cada *instante*, la casilla leída es la adyacente a la derecha del estado actual, el cual se escribe en negritas.

El lector no debe de tener dificultad alguna en reconocer la lista completa de quíntuplas resultante del ejemplo, la cardinalidad del conjunto EA, producto cartesiano del conjunto de estados por el alfabeto, y la representación de la máquina como una lista de tercetas indicada por elementos en el conjunto EA.

1.6.2 Computaciones en máquinas de Turing

Veamos aquí un enfoque formal a las máquinas de Turing.

Una máquina de Turing es una estructura de la forma $M = (Q, A, t, q_0, F)$ donde,

$$\begin{array}{rcl} Q & : & \text{es un conjunto de } \textit{estados}, \\ A & : & \text{es un alfabeto, con un símbolo distinguido, } \textit{blanco}, a_0 \in A, \\ t & : & \text{es una relación de } \textit{transiciones}\text{: } t \subset (Q \times A \times Q \times A \times \textit{Movs}), \\ q_0 & : & \text{es el estado } \textit{inicial}, \text{ y} \\ F & : & \text{es el conjunto de } \textit{estados finales}. \end{array}$$

De manera convencional se dice que

- M es *determinista* si t es una función $t : Q \times A \to Q \times A \times \textit{Movs}$,
- M es *indeterminista* si t es una mera relación,

Una *descripción instantánea* (DI) es una cadena de la forma

$$\mathbf{y}qa\mathbf{x} \equiv \ldots y_{-3}y_{-2}y_{-1}qax_1x_2x_3 \ldots \in A^{-\mathbb{N}} \times Q \times A \times A^{+\mathbb{N}},$$

con $y_{-i} = a_0 = x_j$ c.t.p.-(i, j) (casi en todas partes resepcto a i, j). q es el *estado* de la DI. La interpretación de una tal DI es, naturalmente, la siguiente:

> "El contenido de la cinta es $\mathbf{y}a\mathbf{x}$, se está examinando el carácter a y el control finito de la máquina está en el estado q."

Alternativamente podemos distinguir a un símbolo de marca "$\bullet$" para reconocer inicios y terminaciones de DI's. Una DI es *inicial* o *final* según lo sea su estado. Si $\mathbf{y}q_0a\mathbf{x}$ es inicial decimos que la descripción inicial *corresponde a la entrada* $a\mathbf{x}$. Si $\mathbf{y}q_Fa\mathbf{x}$ es final decimos que la máquina *deja a la salida* $\mathbf{y}$.

Escribimos $\mathbf{y}qa\mathbf{x} \to \mathbf{y}'q'a'\mathbf{x}'$ si es que existe una transición $(q, a, q', b, m) \in t$ tal que

$$\begin{cases} \left\{ \begin{array}{rcl} \mathbf{y}' & = & \mathbf{y}b \\ \mathbf{x} & = & a'\mathbf{x}' \end{array} \right. & \text{si } m = \text{Der} \\ \left\{ \begin{array}{rcl} \mathbf{y} & = & \mathbf{y}'a' \\ \mathbf{x}' & = & b\mathbf{x} \end{array} \right. & \text{si } m = \text{Izq} \end{cases}$$

Denotemos por "$\xrightarrow{*}$" a la cerradura reflexivo-transitiva de la relación "$\to$". Si $q_0\mathbf{x}$ es inicial, $\mathbf{y}q_F$ es final y $q_0\mathbf{x} \xrightarrow{*} \mathbf{y}q_F$ entonces decimos que M *converge* para la entrada $\mathbf{x}$ y escribimos $M(\mathbf{x}) = \mathbf{y}$.

Si $\mathbf{DI} = \{DI_i\}_{i=0,\ldots,k}$ es una sucesión de DI's tal que $DI_0 = q_0\mathbf{x}$, $DI_k = \mathbf{y}q_F$ y $\forall i < k : (DI_i \to DI_{i+1})$ entonces decimos que $\mathbf{DI}$ es una *computación terminal* que transforma la DI inicial en la final. Si DI_k no corresponde a un estado final, la computación se dice ser *intermedia*.

Así pues, si M es una máquina y $\mathbf{x}$ es el contenido inicial en su cinta, $M(\mathbf{x})$ denotará al contenido que queda en la cinta cuando M se para a partir de $\mathbf{x}$. Indicaremos que M no se para a partir de $\mathbf{x}$ escribiendo $M(\mathbf{x}) \uparrow$, en el cual caso, diremos que M *diverge* con $\mathbf{x}$.

1.6.3 Modelos restringidos de máquinas de Turing

Presentaremos algunas variantes de las máquinas de Turing y veremos que son equivalentes todas ellas. Mostraremos las equivalencias entre las diversas modificaciones de las máquinas de Turing de manera más

bien coloquial. Anque se puede presentar demostraciones rigurosas en base a descripciones instantáneas, aquí las omitiremos con la certeza de que el lector podrá construirlas a partir de las indicaciones que presentamos.

$\mathcal{MT}$ denotará a la clase de máquinas de Turing tal como las hemos definido.

Diremos que dos clases de máquinas $\mathcal{M}_1$ y $\mathcal{M}_2$ son *equivalentes* si toda máquina en una clase es *simulada* por una máquina en la otra, es decir,

- $\forall M_1 \in \mathcal{M}_1 \exists M_2 \in \mathcal{M}_2 \forall \mathbf{x} : c(M_1(\mathbf{x})) = M_2(c(\mathbf{x}))$, y
- $\forall M_2 \in \mathcal{M}_2 \exists M_1 \in \mathcal{M}_1 \forall \mathbf{y} : d(M_2(\mathbf{x})) = M_1(d(\mathbf{x}))$,

donde c y d son funciones apropiadas de *conversión* de palabras del alfabeto de una máquina a las del alfabeto de la otra, que cumplen además $c(\uparrow) = \uparrow = d(\uparrow)$, en otras palabras, las conversiones preservan el símbolo de "indefinción".

Máquinas con cintas semi-infinitas, $\mathcal{MTSI}$: Estas son máquinas de Turing donde las casillas de la cinta se ponen en correspondencia con $\mathbb{N}$ mas no con $\mathbb{Z}$, es decir, la cinta de cada máquina de esta clase tiene una casilla inicial a la izquierda, digamos c_0, y se prolonga indefinidamente a la derecha.

Proposición 1.6.1 *$\mathcal{MT}$ y $\mathcal{MTSI}$ son equivalentes.*

Dem. (Bosquejo) Por un lado, tenemos que toda máquina $\mathcal{MTSI}$ es en sí del tipo $\mathcal{MT}$.

Recíprocamente, enumeremos las casillas de la cinta semi-infinita C como $c_0, c_1, \ldots, c_n, \ldots$ Dada una máquina M del tipo $\mathcal{MT}$ enumeremos a las casillas de la cinta de M como $\ldots, d_{-n}, \ldots, d_{-1}, d_0, d_1, \ldots, d_n, \ldots$ Pongamos un símbolo especial S en c_0 e identifiquemos al sector "negativo" de la cinta de M con las casillas impares de C y al "no-negativo" de M con las casillas pares de C:

$$\forall n > 0 : (d_{-n} \leftrightarrow c_{2n-1}) \wedge (d_{n-1} \leftrightarrow c_{2n+2})$$

Observamos que el movimiento hacia una casilla contigua en la cinta de M se traduce a dos movimientos contiguos en C. Cada vez que se "atraviesa" S en C se cambia de sentido: Un movimiento a la derecha en M corresponde a dos hacia la izquierda en C y viceversa.

Hechas estas observaciones, de manera directa se puede simular a M usando C. □

Máquinas con cintas de varias pistas, $\mathcal{MTVP}$: Estas son máquinas de Turing cuyas cintas tienen varias "pistas". Pueden verse también como máquinas de varias cintas con movimientos "sincronizados": Todo movimiento que se haga en una cinta, se hace también en todas.

Proposición 1.6.2 *$\mathcal{MT}$ y $\mathcal{MTVP}$ son equivalentes.*

Dem. (Bosquejo) Por un lado, tenemos que toda máquina $\mathcal{MT}$ es en sí del tipo $\mathcal{MTVP}$.

Recíprocamente, dada una máquina M^k del tipo $\mathcal{MTVP}$ con k pistas, la podemos simular con una máquina M del tipo $\mathcal{MT}$ de cualquiera de las dos maneras alternativas:

1. Si A es el alfabeto de M^k, entonces en cada momento el funcionamiento de M^k depende de la lista de k símbolos que aparece en la casilla actual, consistente de k casillas, una por cada pista. Esto define a una máquina del del tipo $\mathcal{MT}$ sobre el alfabeto A^k que es una potencia cartesiana del original.
2. Consideremos una cinta de una sola pista en donde sus casillas se agrupan en bloques de k casillas. La posición j-ésima de la i-ésima pista en la cinta de M^k corresponde a la posición $((j-1)\cdot k + i)$-ésima de la nueva cinta. En esta nueva, casillas contiguas en la misma pista de M^k distanciarán de k casillas. El mecanismo de funcionamiento se modifica para que cada movimiento de M^k obligue a revisar un bloque de k casillas y haga la modificación correspondiente, trabajando siempre en bloques. □

Máquinas con varias cintas, $\mathcal{MTVC}$: Estas son máquinas de Turing con varias cintas con movimientos "independientes".

Proposición 1.6.3 *$\mathcal{MT}$ y $\mathcal{MTVP}$ son equivalentes.*

DEM. (BOSQUEJO) Demostraremos que $\mathcal{MTVP}$ y $\mathcal{MTVC}$ son equivalentes.

En efecto, por un lado, tenemos que toda máquina $\mathcal{MTVP}$ es en sí del tipo $\mathcal{MTVC}$.

Recíprocamente, dada una máquina M^{ind} del tipo $\mathcal{MTVC}$ con k pistas, la podemos simular con una máquina M^{sinc} del tipo $\mathcal{MTVP}$, con $2k$ pistas como sigue:

Cada cinta C_j da origen a dos pistas P_{j0}, P_{j1}. La pista P_{j1} tiene el mismo contenido que la cinta C_j. La pista P_{j0} está en blanco salvo en la posición donde se encuentra la j-ésima cabeza lectora de M^{ind}, en la cual hay una marca "$\downarrow$" para marcar que ésa es la casilla escudriñada en la j-ésima cinta.

Con esto, la máquina M^{sinc} se construye de manera inmediata. □

Máquinas con dos símbolos, $\mathcal{MT}01$: Estas son máquinas de Turing sobre el alfabeto $(0+1)$.

Proposición 1.6.4 *$\mathcal{MT}$ y $\mathcal{MT}01$ son equivalentes.*

DEM. (BOSQUEJO) Por un lado, tenemos que toda máquina $\mathcal{MT}01$ es en sí del tipo $\mathcal{MT}$.

Recíprocamente, dada una máquina M del tipo $\mathcal{MT}$ sobre un alfabeto con m símbolos, sea $k = \lceil \log_2 m \rceil$. Cada símbolo en el alfabeto puede codificarse mediante una cadena de k bits. Por tanto M puede verse como una m"aquina de k pistas. De acuerdo con la segunda construcción de la máquina del tipo $\mathcal{MT}$ simuladora de una máquina del tipo $\mathcal{MTVP}$, obtenemos una máquina del tipo $\mathcal{MT}01$ que simula a M. □

1.6.4 Computabilidad-Turing

Una función $\mathbb{N}^n \to \mathbb{N}$ se dice ser *computable-T*, o *computable-Turing*, si existe una máquina de Turing que la calcula.

Aunque básicamente es correcta esta definición, es evidente que quedan algunos elementos en ella que aún no han sido precisados.

Antes que nada, consideraremos en esta sección que los números naturales se están representando en base 2, utilizando únicamente a los dígitos $0, 1$. Consideraremos también al menos dos símbolos más: la coma "," para separar números, y el "blanco", "b", para marcar en su totalidad, salvo un número finito de casillas, a la cinta de cualquier máquina de Turing. Así, por ejemplo, el vector $(1, 2, 3)$ se representa por la cadena $b1, 10, 11b$. Sea $A_0 = \{0, 1, ', , b\}$ el alfabeto consistente de esos cuatro símbolos.

Dado $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{N}^n$ la representación de $\mathbf{x}$ es $\lceil \mathbf{x} \rceil =' b(x_1)_2, \cdots, (x_n)_2 b'$ donde para cada número $m \in \mathbb{N}$, $(m)_2$ es la representación binaria de m. Aunque $\mathbf{x}$ y $\lceil \mathbf{x} \rceil$ son dos entes distintos (uno es un objeto, un vector, y el otro es la representación de ese objeto), en la práctica los confundiremos y no haremos más distinción entre objetos y sus representaciones.

Sea A un alfabeto que contenga a A_0 y sea M una máquina de Turing sobre A con estado inicial q_0. Sea $\mathbf{x} \in \mathbb{N}^n$. Por $q_0\mathbf{x}$ denotamos a la *descripción instantánea* que dice que el contenido de la cinta de la máquina de Turing es la representación del vector $\mathbf{x}$, que el estado actual de la máquina es el inicial q_0 y que se está escudriñando la casilla más a la izquierda de $\lceil \mathbf{x} \rceil$ que no está en blanco. Si q es un estado final, con la notación $\mathbf{y}q$ designaremos la situación de que la máquina ha arribado al estado q y la cadena de 0's, 1's y comas inmediatamente a la izquierda de la casilla escudriñada corresponde a un vector de números naturales.

Escribiremos $M \vdash q_0\mathbf{x} \overset{*}{\rightarrow} \mathbf{y}q$ para indicar que mediante computaciones legales de M, partiendo de la descripción instantánea $q_0\mathbf{x}$, en algún momento se deriva la descripción $\mathbf{y}q$. En tal caso, escribimos $M(\mathbf{x}) = \mathbf{y}$. Si la máquina no se para a partir de $q_0\mathbf{x}$ escribiremos $M(\mathbf{x}) \uparrow$. La función que *calcula* M es $f_M : \mathbf{x} \mapsto M(\mathbf{x})$.

Así pues, una función $f : \mathbb{N}^n \rightarrow \mathbb{N}$ es computable-T si y sólo si existe una máquina de Turing sobre un alfabeto $A \supset A_0$ tal que $f = f_M$.

Comparemos pues las nociones de computabilidad introducidas hasta ahora:

Lema 1.6.1 *Toda función computable-T es computable (mediante programas-***while***).*

Dem. (Bosquejo) La demostración formal de este lema puede ser muy engorrosa y acaso poco ilustrativa. Lo que hay que ver, en esencia, es que el programa de cualquier máquina de Turing, es decir, su mecanismo de control visto como una lista de quíntuplas, puede ser convertido en un programa-**while**. En otras palabras, lo que hay que ver es que toda máquina de Turing puede ser simulada por un programa-**while**. Lo que no es otra cosa sino que las máquinas de Turing pueden ser programables en el marco de la programación-**while**. Cualquier programador con experiencia media quedará convencido de que esto último, en efecto, sí es posible. □

Recíprocamente, se tiene

Lema 1.6.2 *Toda función computable es computable-T.*

Dem. Observemos las reglas que definen a los programas-**while** en la figura (1.6). Primeramente observamos que el decidir si acaso el contenido de una variable es nulo o no lo podemos hacer con máquinas de Turing. También mediante ellas podemos calcular a las funciones $x++$ y $x--$.

Ahora es claro que dadas dos máquinas de Turing M_1 y M_2, se puede construir una tercera máquina de Turing M_3 tal que, para cualquier posible entrada $\mathbf{x}$ se tiene $M_3(\mathbf{x}) = M_2(M_1(\mathbf{x}))$. Por tanto, si los programas-**while** P_1 y P_2 fueran simulados por las máquinas de Turing M_1 y M_2 entonces la concatenación $P_1; P_2$ se simula por M_3. En otras palabras, la clase de máquinas de Turing es cerrada bajo *composición.*

También, tenemos que si el programa-**while** P_1 fuese simulado por una máquina de Turing M entonces el esquema $\boxed{\textbf{while } x \neq 0 \textbf{ do } P_1}$ se puede también simular por una máquina de Turing. □

De ambos lemas, obtenemos,

Teorema 1.6.1 *Las nociones de computabilidad-Turing y computabilidad (mediante programas-***while***) coinciden.*

1.7 Tesis de Church

Tesis de Church: El concepto anterior de *computabilidad* coincide con el intuitivo de *computabilidad efectiva.*

En el presente curso formularemos varios enfoques alternativos de la noción de *computabilidad.* Siendo que todos esos son equivalentes, la tesis de Church ha prevalecido. No hay manera de demostrarla pues es tan solo una convención.

1.8 Propiedades de cerradura de funciones computables

En la sección anterior introdujimos a las funciones computables como aquellas que son calculables por programas-**while**. En esta sección presentaremos algunas propiedades de cerradura de las funciones computables, vale decir, presentaremos algunos esquemas de composición de funciones tales que actuando sobre funciones computables dan, también, funciones computables. Veremos además que, aunque la clase de funciones computables es muy grande, es numerable. Esto, con un simple argumento de cardinalidad de conjuntos, implica que hay una gran cantidad de funciones de los naturales en los naturales que no son computables.

Si A y B son dos conjuntos, escribiremos

$$B^A = \{f : A \to B | f \text{ es una función}\}$$

para denotar a las funciones con dominio en A y contradominio en B.

Esquema de composición

Este esquema proporciona la composición típica de funciones, es decir, la composición en el sentido algebraico.

Dada una función de k argumentos y dadas k funciones, de n argumentos cada una, al componer a las segundas con la primera obtenemos una función de n argumentos,

$$\boxed{Comp : (\mathbb{N})^{\mathbb{N}^k} \times \left((\mathbb{N})^{\mathbb{N}^n}\right)^k \to (\mathbb{N})^{\mathbb{N}^n}.}$$

Si $f_1, \ldots, f_k : \mathbb{N}^n \to \mathbb{N}$ son k funciones y $g : \mathbb{N}^k \to \mathbb{N}$ es otra función, la *composición* de las f's con la g es la función

$$\begin{array}{rcl} g(f_1, \ldots, f_k) : \mathbb{N}^n & \to & \mathbb{N} \\ \mathbf{x} & \mapsto & g(f_1(\mathbf{x}), \ldots, f_k(\mathbf{x})) \end{array}$$

Hacemos, $Comp(g; f_1, \ldots, f_k) = g(f_1, \ldots, f_k)$.

Esquema de minimización

Dada una función $f : (\mathbf{x}, y) \mapsto f(\mathbf{x}, y)$ de $n+1$ argumentos podemos construir una nueva función $g : \mathbf{x} \mapsto g(\mathbf{x})$ asociándole a cada $\mathbf{x}$ el mínimo y para el cual se anula la *sección* $f_{\mathbf{x}} : y \mapsto f(\mathbf{x}, y)$ de f a la altura $\mathbf{x}$. g se obtiene de *minimizar* a f. Así pues, este esquema toma una función de $n+1$ argumentos y produce otra de n argumentos,

$$\boxed{Mini : (\mathbb{N})^{\mathbb{N}^{n+1}} \to (\mathbb{N})^{\mathbb{N}^n}.}$$

Más precisamente:

Si $f : \mathbb{N}^{n+1} \to \mathbb{N}$ es una función *total*, es decir, definida en todos las posibles instancias a sus argumentos, definimos

$$\begin{array}{rcl} (\mu_y[f = 0]) : \mathbb{N}^n & \to & \mathbb{N} \\ \mathbf{x} & \mapsto & \begin{cases} \text{Min}\{y | f(\mathbf{x}, y) = 0\} & \text{si existe tal } y, \\ \perp & \text{en otro caso.} \end{cases} \end{array}$$

Hacemos, $Mini(f) = (\mu_y[f = 0])$.

Esquema de minimización acotada

La única diferencia, que efectivamente es muy sustancial, con el esquema de minimización es que dada una función $f : (\mathbf{x}, y) \mapsto f(\mathbf{x}, y)$ de $n+1$ argumentos al construir su minimización $g : \mathbf{x} \mapsto g(\mathbf{x})$ asociándole a cada $\mathbf{x}$ el mínimo y para el cual se anula la sección $f_{\mathbf{x}} : y \mapsto f(\mathbf{x}, y)$, para cada $\mathbf{x}$ se busca la y correspondiente hasta que no se exceda un cierto límite z marcado por un $(n+1)$-ésimo argumento para g. Así pues, este esquema toma una función de $n+1$ argumentos y produce otra de $n+1$ argumentos,

$$\boxed{MiAc : (\mathbb{N})^{\mathbb{N}^{n+1}} \to (\mathbb{N})^{\mathbb{N}^{n+1}}.}$$

Si $f : \mathbb{N}^{n+1} \to \mathbb{N}$ es una función cualquiera definimos

$$\begin{array}{rcl} (\mu_{y\leq z}[f=0]) : \mathbb{N}^n \times \mathbb{N} & \to & \mathbb{N} \\ (\mathbf{x}, z) & \mapsto & \begin{cases} \text{Min}\{y \leq z | f(\mathbf{x}, y) = 0\} & \text{si existe tal } y, \\ z & \text{en otro caso.} \end{cases} \end{array}$$

Observamos que si $\mu_{y\leq z}[f=0](\mathbf{x}, z) = z$ entonces pueden ocurrir dos situaciones: Bien $f(\mathbf{x}, z) = 0$ o bien ningún $y \leq z$ cumple que $f(\mathbf{x}, z) = 0$. Cuál de estos dos casos ocurre puede dilucidarse calculando el valor de $\mu_{y\leq z+1}[f=0](\mathbf{x}, z+1)$.

Hacemos, $MiAc(f) = (\mu_{y\leq z}[f=0])$.

Esquema de recursión

Una típica aplicación de este esquema define a una función f haciendo

$$\boxed{f(\mathbf{x}, y) := [\textbf{if } y = 0 \textbf{ then } g(\mathbf{x}) \textbf{ else } h(\mathbf{x}, f(\mathbf{x}, y-1))]}$$

La función g está definiendo el caso "*base*" de la recurrencia, en tanto que la función h define precisamente el *recurrente*. En h, puede también ocurrir y como un argumento. Así pues, este esquema se aplica sobre dos funciones, la primera correspondiente al caso "base" con $n-1$ argumentos y la segunda correspondiente al caso recurrente con $n+1$ argumentos, y produce una función con n argumentos,

$$\boxed{Recu : (\mathbb{N})^{\mathbb{N}^{n-1}} \times (\mathbb{N})^{\mathbb{N}^{n+1}} \to (\mathbb{N})^{\mathbb{N}^n}.}$$

Más precisamente:

Si $g : \mathbb{N}^{n-1} \to \mathbb{N}$ y $h : \mathbb{N}^{n+1} \to \mathbb{N}$ son dos funciones definimos

$$\begin{array}{rcl} f : \mathbb{N}^n & \to & \mathbb{N} \\ (\mathbf{x}, y) & \mapsto & f(\mathbf{x}, y) \end{array}$$

donde

$$\begin{array}{lrcl} \textit{Caso base}: & f(\mathbf{x}, 0) & = & g(\mathbf{x}) \\ \textit{Caso recurrente}: \forall y & f(\mathbf{x}, y+1) & = & h(\mathbf{x}, y, f(\mathbf{x}, y)) \end{array}$$

Hacemos, $Recu(g; h) = f$.

Esquema de recursión acotada

Este esquema es similar al anterior, con la sola excepción de que los valores de la función que se define recurrentemente no excedan los de una tercera función dada,

Input: k functions $f_1, \ldots, f_k : \mathbb{N}^n \to \mathbb{N}$,
$g : \mathbb{N}^k \to \mathbb{N}$ and
$\mathbf{x} \in \mathbb{N}^n$
Output: $r = Comp(g; f_1, \ldots, f_k)(\mathbf{x})$

{ **for** $i = 1$ **to** k **do** $y_i := f_i(\mathbf{x})$;
$r := g(y_1, \ldots, y_k)$
}

Figura 1.7: Seudocódigo correspondiente al Esquema de Composición.

$$ReAc : (\mathbb{N})^{\mathbb{N}^{n-1}} \times (\mathbb{N})^{\mathbb{N}^n} \times (\mathbb{N})^{\mathbb{N}^{n+1}} \to (\mathbb{N})^{\mathbb{N}^n} .$$

Más precisamente:

Si $g : \mathbb{N}^{n-1} \to \mathbb{N}$, $h : \mathbb{N}^{n+1} \to \mathbb{N}$ y $c : \mathbb{N}^n \to \mathbb{N}$ son tres funciones definimos

$$\begin{array}{rcl} f : \mathbb{N}^n & \to & \mathbb{N} \\ (\mathbf{x}, y) & \mapsto & f(\mathbf{x}, y) \end{array}$$

donde

$$\begin{array}{lrcl} \textit{Caso base} : & f(0, \mathbf{x}) & = & g(\mathbf{x}) \\ \textit{Caso recurrente} : \forall y & f(\mathbf{x}, y+1) & = & h(\mathbf{x}, y, f(\mathbf{x}, y)) \\ \textit{Condición de acotamiento} : \forall y & f(\mathbf{x}, y) & \leq & c(\mathbf{x}, y) \end{array}$$

Hacemos, $ReAc(g; h; c) = f$.

Computabilidad de los esquemas

Proposición 1.8.1 *Si $g, f_1, \ldots, f_k, f, h, c$ son computables, entonces*

$$\begin{array}{lllll} Comp(g; f_1, \ldots, f_k) & , & & & \\ Mini(f) & , & MiAc(f) & , & \\ Recu(g; h) & \text{y} & ReAc(g; h; c) & . & \end{array}$$

son computables también.

Dem. (Bosquejo) En la Figura 1.7 presentamos un seudocódigo para calcular la composición de funciones. En la Figura 1.8 presentamos sendos seudocódigos para calcular el esquema de minimización (incondicional) y el correspondiente acotado. En la Figura 1.9 presentamos sendos seudocódigos recursivos para calcular el esquema de recursión (incondicional) y el correspondiente acotado, sin embargo estos dos últimos seudocódigos no son estrictamente "legales" pues no hemos aún introducido la noción de *programa*-**while** *recursivo*. En la Figura 1.10 presentamos otros dos seudocódigos para los esquemas de recursión utilizando únicamente macros ya introducidos. A partir de tales ejemplos el lector podrá observar la transformación que se ha de hacer sobre los programas recursivos para obtener programas-**while** equivalentes. □

1.9 Numerabilidad de la clase de funciones computables

Veremos en esta sección que la clase de los programs-**while** es numerable.

Input: $f : \mathbb{N}^{n+1} \to \mathbb{N}$ total function and $\mathbf{x} \in \mathbb{N}^n$
Output: $r = Mini(f)(\mathbf{x})$

$\{ \quad y := 0 \ ;$
while $f(\mathbf{x}, y) \neq 0$ **do** $y++ \ ;$
$r := y \ \}$

Input: $f : \mathbb{N}^{n+1} \to \mathbb{N}$ total function and $(\mathbf{x}, z) \in \mathbb{N}^{n+1}$
Output: $r = MiAc(f)(\mathbf{x}, z)$

$\{ \quad y := 0 \ ;$
while $[f(\mathbf{x}, y) \neq 0] \wedge [y < z]$ **do** $y++ \ ;$
$r := y \ \}$

Figura 1.8: Seudocódigos correspondientes a los Esquemas de Minimización.

Input: $g : \mathbb{N}^{n-1} \to \mathbb{N}$ base function, $h : \mathbb{N}^{n+1} \to \mathbb{N}$ recurrent function and $(\mathbf{x}, y) \in \mathbb{N}^n$
Output: $r = Recu(g; h)(\mathbf{x}, y)$

```
Algorithm Recu
{  if y = 0 then r := g(x)
   else
   {  w := Recu(g, h, (x, y − 1)) ;
      r := h(x, y, w)
   }
}
```

Input: $g : \mathbb{N}^{n-1} \to \mathbb{N}$ base function, $h : \mathbb{N}^{n+1} \to \mathbb{N}$ recurrent function, $c : \mathbb{N}^n \to \mathbb{N}$ bounding function and $(\mathbf{x}, y) \in \mathbb{N}^n$
Output: $r = ReAc(g; h)(\mathbf{x}, y)$

```
Algorithm ReAc
{  if [y = 0] ∧ [g(x) ≤ c(x)] then r := g(x)
   else
   {  w := ReAc(g, h, (x, y − 1)) ;
      r1 := h(x, y, w) ;
      if r1 ≤ c(x, y) then r := r1
   }
}
```

Figura 1.9: Seudocódigos recursivos correspondientes a los Esquemas de Recursión.

Input: $g : \mathbb{N}^{n-1} \to \mathbb{N}$ base function, $h : \mathbb{N}^{n+1} \to \mathbb{N}$ recurrent function and $(\mathbf{x}, y) \in \mathbb{N}^n$
Output: $r = Recu(g; h)(\mathbf{x}, y)$

$\{ \quad y_0 = 0 \ ; \ r_0 := g(\mathbf{x}) \ ;$
while $y_0 < y$ **do**
$\{ \ r_0 := h(\mathbf{x}, y_0 + 1, r_0) \ ; \ y_0++ \ \}$
$r := r_0$
$\}$

Input: $g : \mathbb{N}^{n-1} \to \mathbb{N}$ base function, $h : \mathbb{N}^{n+1} \to \mathbb{N}$ recurrent function, $c : \mathbb{N}^n \to \mathbb{N}$ bounding function and $(\mathbf{x}, y) \in \mathbb{N}^n$
Output: $r = ReAc(g; h)(\mathbf{x}, y)$

$\{ \quad y_0 = 0 \ ; \ r_0 := g(\mathbf{x}) \ ;$
while $y_0 < y$ **do**
$\{ \ r_0 := h(\mathbf{x}, y_0 + 1, r_0) \ ; \ y_0++ \ \}$
if $r_0 \leq c(\mathbf{x}, y)$ **then** $r := r_0$
$\}$

Figura 1.10: Seudoprogramas-**while** correspondientes a los Esquemas de Recursión.

0	r_0	$\equiv$	**while** $x \neq 0$ **do** $x--$	p_0	$\equiv$	$\{r_0\}$
1	r_1	$\equiv$	$r_0; x++$	p_1	$\equiv$	$\{r_1\}$
2	r_2	$\equiv$	$r_1; x++$	p_2	$\equiv$	$\{r_2\}$
$\vdots$		$\vdots$			$\vdots$	
$\mathbf{n+1}$	r_{n+1}	$\equiv$	$r_n; x++$	p_{n+1}	$\equiv$	$\{r_{n+1}\}$
$\vdots$		$\vdots$			$\vdots$	

En cada renglón, en la primera columna aparece el correspondiente natural, en la segunda la lista de instrucciones y en la tercera el numeral correspondiente, visto éste como un programa-**while**.

Tabla 1.6: Numerales como programas-**while**.

Un *numeral* es la representación formal de un número natural en una teoría dada. Así, por ejemplo, la cadena de $(n+1)$ 1's, 1^{n+1}, es el numeral del número n en la teoría 1^* (tomamos como 1 al numeral de 0 para distinguir a 0 de la palabra vacía). La representación en base 10 de un número natural n puede ser vista como el numeral de n en el lenguaje $(0+1+2+3+4+5+6+7+8+9)^*$. Se sabe bien que en este último lenguaje, el numeral de un número es único salvo 0's a la izquierda, "pues ésos no cuentan".

Observación 1.9.1 *Los número naturales son expresables mediante numerales constructibles como programas-***while**.

En la tabla 1.6 presentamos una descripción inductiva de esos numerales.

Esta propiedad de los programas-**while** hace que los programas-**while** se puedan codificar mediante ellos mismos, simplemente componiendo las correspondencias siguientes:

$$\begin{array}{ccccccc} \boxed{\text{Clase de programas-}\mathbf{while}} & \xrightarrow{\Phi} & \mathbb{N} & \xrightarrow{\Psi} & \boxed{\text{Clase de programas-}\mathbf{while}} \\ P & \mapsto & n = \Phi(P) & \mapsto & p_n \end{array}$$

donde Φ es la función de enumeración de los programas-**while**, de cuya existencia nos ocuparemos inmediatamente, y Ψ es la correspondencia entre números y numerales descrita en la tabla 1.6.

1.9.1 Codificación de programas-while

Escritura de variables

Fijemos primero un acuerdo para utilizar variables en los programas-**while**. Supondremos a las variables formadas por el prefijo X y una cadena en $\{0,1\}$. Así pues, las variables son las cadenas formadas de acuerdo con las siguientes reglas gramaticales:

$$\begin{array}{rcl} \langle \textit{Dígito} \rangle & ::= & 0|1 \\ \langle \textit{ListaDígitos} \rangle & ::= & \textit{nil}|\langle \textit{Dígito} \rangle \langle \textit{ListaDígitos} \rangle \\ \langle \textit{Variable} \rangle & ::= & X \langle \textit{ListaDígitos} \rangle \end{array}$$

Codificación de símbolos

El alfabeto de los programas-**while** consta de 11 símbolos. Puesto que $2^3 < 11 < 2^4$ utilizaremos 4 bits para referirnos a cada uno de los símbolos y utilizaremos un bit más como "separador" de símbolos. En la tabla 1.7 presentamos la codificación de los símbolos de los programas-**while** que utilizaremos en esta sección.

Símbolo	Código binario	Código decimal
while	10000	16
do	10001	17
{	10010	18
}	10011	19
$++$	10100	20
$--$	10101	21
$\neq$	10110	22
X	10111	23
0	11000	24
1	11001	25
;	11010	26

Tabla 1.7: Codificación del alfabeto de los programas-**while**.

Codificación de programas y tiras de símbolos

Cada palabra P, en el alfabeto de los programas-**while**, se codifica por el número cuya representación binaria coincide con la yuxtaposición de los códigos de los símbolos en las palabras. Esto define una función

$$\lceil\cdot\rceil : \text{programas-}\mathbf{while} \to \mathbb{N}.$$

Ejemplos:

1. $\lceil ; X++\rceil = (11010\ 10111\ 10100)_2 = 27380.$

2. $$\begin{aligned}\lceil r_0\rceil &= \lceil \mathbf{while}\ x \neq 0\ \mathbf{do}\ x--\rceil \\ &= (10000\ 10111\ 10110\ 11000\ 10001\ 10111\ 10101)_2 \\ &= 17975494389\end{aligned}$$

3. $$\begin{aligned}\lceil \mathbf{0}\rceil &= \lceil p_0\rceil \\ &= 18\cdot 32^8 + \lceil r_0\rceil \cdot 32 + 19 \\ &= 1193691111091\end{aligned}$$

4. De manera recursiva, tenemos $\forall n > 0$:

$$\begin{aligned}\lceil r_n\rceil &= \lceil r_{n-1}\rceil \cdot 32^3 + \lceil ; X++\rceil \\ &= 32^{3n}\lceil r_0\rceil + \lceil ; X++\rceil \frac{32^{3n}-1}{32^3-1} \\ &= \frac{1}{32767}\left(589003024671743\cdot 32^{3n} - 27380\right)\end{aligned}$$

y

$$\begin{aligned}\lceil \mathbf{n}\rceil &= \lceil p_n\rceil \\ &= 18\cdot 32^{8+3n} + \lceil r_n\rceil \cdot 32 + 19\end{aligned}$$

Así se tiene que, de acuerdo con la codificación indicada, los códigos de los primeros 6 numerales son:

$$\begin{aligned}\lceil \mathbf{0}\rceil &= 1193691111091 \\ \lceil \mathbf{1}\rceil &= 667367018346667667 \\ \lceil \mathbf{2}\rceil &= 21868282457183606365843 \\ \lceil \mathbf{3}\rceil &= 716579879556992413396197011 \\ \lceil \mathbf{4}\rceil &= 23480889493323527402166583910035 \\ \lceil \mathbf{5}\rceil &= 769421786917225345914194621564280467\end{aligned}$$

Observaciones sobre la codificación

Denotemos por Alf^* al conjunto de tiras sobre el alfabeto de los programas-**while**.

1. $\lceil\cdot\rceil : Alf^* \to \mathbb{N}$ es una función
   (a) efectivamente calculable, es decir, dada una tira cualquiera $P \in Alf^*$ se obtiene de manera procedimental su código $\lceil P\rceil$,
   (b) inyectiva, pues para que $\lceil P_1\rceil = \lceil P_2\rceil$ es necesario que $P_1 = P_2$, cualesquiera que sean las tiras $P_1, P_2 \in Alf^*$, y
   (c) que no es suprayectiva, pues la representación binaria de cualquier número en la imagen de $\lceil\cdot\rceil$ ha de tener una longitud que es un múltiplo de 5 y en cada posición múltiplo de 5, contada de derecha a izquierda, ha de estar "prendido" el bit correspondiente. En otras palabras, un número está en la imagen de $\lceil\cdot\rceil$ si y sólo si al escribir a ese número en base $32 = 2^5$, sólo aparecen los dígitos que están entre los números decimales 16 y 26 inclusive.
2. Puesto que la clase de los programas-**while** es un subconjunto propio de Alf^* tenemos que el conjunto de códigos de programas-**while**, $N \subset \mathbb{N}$, es un subconjunto propio de la imagen de $\lceil\cdot\rceil$.
3. Sin embargo, dado $n \in \mathbb{N}$ es algorítmicamente decidible si acaso existe un programa-**while** P tal que $n = \lceil P\rceil$.

   En efecto, dado n se puede calcular, mediante la "división larga" de enteros, la representación en base 32 de n y, obtenida ésta, hacer la conversión a una palabra $p_n \in Alf^*$ mediante la codificación descrita en la tabla 1.7. Luego, mediante un autómata de pila, se puede ver si acaso p_n se ajusta a la gramática de los programas-**while**.
4. El procedimiento descrito en el punto anterior para obtener $p_n \in Alf^*$ dado $n \in \mathbb{N}$ es un procedimiento efectivo para calcular la inversa de la codificación $\lceil\cdot\rceil$.
5. La codificación Φ mencionada inmediatamente después de la observación 1.9.1 es, precisamente, la codificación $\lceil\cdot\rceil$ restringida a la clase de programas-**while**.

Para cada programa-**while** P, el número $i_P = \lceil P\rceil$ es el *índice* del programa P. Consecuentemente, P se dice ser el i_P*-ésimo programa*-**while** $P = P_{i_P}$.

Ahora bien, las funciones computables son las que se calculan mediante programas-**while**. Por tanto, podemos generalizar la noción de *índice* a las funciones computables como sigue:

$\forall n \geq 0$ y para cada $i \in N$, el i-ésimo programa-**while** P_i calcula a la función

$$\begin{array}{rcl} f_i^{(n)} : \mathbb{N}^n & \to & \mathbb{N} \\ (x_1, \ldots, x_n) & \mapsto & y = f_i^{(n)}(x_1, \ldots, x_n) \end{array}$$

donde y es el valor obtenido como sigue:

1. Si $X_1, \ldots, X_k$ es la lista de variables de P_i, en función de cómo se compare n con k, se instanciará a las variables de entrada como sigue:

$$k \leq n : \forall j \leq k(X_j := x_j)$$

$$k > n : [\forall j \leq n(X_j := x_j)] \wedge [\forall j \in [\![n+1, k]\!](X_j := 0)].$$

2. El valor que queda en la última variable X_k cuando termina P_i, si acaso terminare, es el asignado a y.

Una función es computable si coincide con alguna de la forma $f_i^{(n)}$. Si $f : \mathbb{N}^n \to \mathbb{N}$ es computable y $f = f_i^{(n)}$ entonces i se dice ser un *índice* de f.

Proposición 1.9.1 *Toda función computable tiene una infinidad de índices.*

DEM. (BOSQUEJO) Sin entrar en minucias, tenemos que dos programas-**while** calculan a una misma función si, por ejemplo, uno posee variables "mudas", es decir, irrelevantes para el cálculo, que no aparecen en el otro, o bien si uno posee "instrucciones ociosas", digamos una secuencia de la forma $(x++; x--;)^*$ y, en todo lo demás, coincide con el otro programa. Dos tales programas son equivalentes y cada tal programa da un índice para la función que calcula. □

Capítulo 2

Funciones recursivas primitivas

En este capítulo haremos una presentación formal de las funciones recursivas primitivas. Veremos que todas ellas son computables.

Introduciremos la jerarquía de Grzegorczyk como un ejemplo de una jerarquía de complejidad. En el primer nivel de esta jerarquía están las así llamadas funciones elementales. Mostraremos que éstas pueden ser definidas, siempre de manera equivalente, partiendo de diversos esquemas de composición. Finalmente, la función de Ackermann nos proveerá de un ejemplo de una función que, siendo computable, no es recursiva primitiva.

2.1 Conceptos básicos

Recordamos que $\mathbb{N}^*$ denota al conjunto de secuencias de números naturales, incluída entre ellas a la vacía, *nil*, y que este conjunto es numerable. Por tanto, módulo una función biyectiva podemos identificar a $\mathbb{N}^*$ con $\mathbb{N}$ mismo, y lo mismo podría hacerse para cada producto cartesiano $\mathbb{N}^n$, con $n > 0$.

De manera estricta, podríamos plantear a la teoría de las funciones recursivas primitivas, considerando a todas ellas con dominio en $\mathbb{N}$. Sin embargo, es convencional referirse a un dominio del tipo $\mathbb{N}^n$ para cada función recursiva primitiva y, así, nos ajustaremos a esa convención.

La clase $\mathcal{FRP}$ de las *funciones recursivas primitivas* es la clase mínima de funciones que cumple con las propiedades siguientes:

1. *Caso "base".* Las siguientes funciones están en la clase $\mathcal{FRP}$:

Función "Cero"	.	$0 : \mathbb{N} \to \mathbb{N}$	,	$x \mapsto 0$.
Función "Sucesor"	.	$suc : \mathbb{N} \to \mathbb{N}$	,	$x \mapsto x + 1$.
i-ésima proyección de orden n	.	$\pi_i^n : \mathbb{N}^n \to \mathbb{N}$	,	$\mathbf{x} = (x_1, \ldots, x_n) \mapsto x_i$.

2. *Caso inductivo.* La clase $\mathcal{FRP}$ es cerrada bajo los esquemas de composición y recursión, i.e.

$$\begin{aligned} g, f_1, \ldots, f_k \in \mathcal{FRP} &\Rightarrow Comp(g; f_1, \ldots, f_k) \in \mathcal{FRP} \\ g, h \in \mathcal{FRP} &\Rightarrow Recu(g; h) \in \mathcal{FRP} \end{aligned}$$

Proposición 2.1.1 *Toda función recuriva primitiva es computable.*

Dem. Se puede mostrar esto por inducción en la definición de las funciones recursivas primitivas. Para esto, basta ver que cada una de las funciones básicas en las recursivas primitivas es computable, lo cual es directo;

y hay que tener en cuenta que la clase de las funciones computables es cerrada bajo los dos esquemas de composición y de recursión que definen a las recursivas primitivas. □

Veamos algunos ejemplos de funciones en $\mathcal{FRP}$.

1. *Constantes.* Para cualquier $m \in \mathbb{N}$ la función constante $x \mapsto m$ está en $\mathcal{FRP}$.

   En efecto, probemos la aseveración por inducción en m.

   Para $m = 0$ se cumple pues la función Cero está efectivamente en $\mathcal{FRP}$.

   Sea $m > 0$. Supongamos que la función $c_m : x \mapsto m$ está en $\mathcal{FRP}$. Es claro que $c_{m+1} = suc \circ c_m$. Por tanto, c_{m+1} está en $\mathcal{FRP}$ como composición de dos funciones en $\mathcal{FRP}$.

2. *Suma.* Ya que para cualesquiera $x, y \in \mathbb{N}$ se cumple

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

   se tiene que

   $$\begin{aligned} Suma(x, 0) &= identidad(x) \\ Suma(x, suc(y)) &= suc(Suma(x, y)) \end{aligned}$$

   Por tanto, $Suma(x, y) = Recu(\pi^1_1, h)(x, y)$, donde $\forall x, y, z : h(x, y, z) = suc \circ \pi^3_3(x, y, z)$, de lo cual se sigue directamente que $Suma \in \mathcal{FRP}$ pues se obtiene por el esquema de recursión de funciones en $\mathcal{FRP}$.

3. *Producto.* Ya que para cualesquiera $x, y \in \mathbb{N}$ se cumple

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

   se tiene que

   $$\begin{aligned} Producto(x, 0) &= 0 \\ Producto(x, suc(y)) &= Suma(Producto(x, y), identidad(x)) \end{aligned}$$

   Por tanto, $Producto(x, y) = Recu(0, h)(x, y)$, donde $\forall x, y, z : h(x, y, z) = Suma(\pi^3_1, \pi^3_3)(x, y, z)$, de lo cual se sigue directamente que $Producto \in \mathcal{FRP}$ pues se obtiene por el esquema de recursión de funciones en $\mathcal{FRP}$.

4. *Suma reiterada.* Si f es una función recursiva primitiva definamos $g : y \mapsto \sum_{x=0}^{y} f(x)$. g es la *sumatoria de* f.

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

   se tiene que

   $$\begin{aligned} g(0) &= f(0) \\ g(suc(y)) &= Suma(f(suc(y)), g(y)) \end{aligned}$$

   Por tanto, $g(y) = Recu(f \circ \pi^1_1, h)(y)$, donde $\forall y, z : h(y, z) = Suma(f(suc \circ \pi^2_1), \pi^1_2)(y, z)$, de lo cual se sigue directamente que $g \in \mathcal{FRP}$ pues se obtiene por el esquema de recursión de funciones en $\mathcal{FRP}$.

   Lo mismo vale si, en vez de considerar a la operación *Suma*, se considera a una operación binaria que sea asociativa, como lo es, como un segundo ejemplo, la operación *Producto*.

5. *Signo.* Sea $sgn : x \mapsto \begin{cases} 0 & \text{si } x = 0, \\ 1 & \text{si } x > 0. \end{cases}$.

   Tenemos

$$\begin{aligned} sgn(0) &= 0 \\ sgn(y+1) &= 1 \end{aligned}$$

   por lo que se tiene

$$\begin{aligned} sgn(0) &= 0 \\ sgn(suc(y)) &= suc(0) \end{aligned}$$

   Por tanto, $sgn(y) = Recu(0, (suc \circ 0))(y)$, de lo cual se sigue directamente que $sgn \in \mathcal{FRP}$ pues se obtiene por el esquema de recursión de funciones en $\mathcal{FRP}$.

6. *Signo complementado.* Sea $\overline{sgn} : x \mapsto \begin{cases} 1 & \text{si } x = 0, \\ 0 & \text{si } x > 0. \end{cases}$.

   Tenemos

$$\begin{aligned} \overline{sgn}(0) &= 1 \\ \overline{sgn}(y+1) &= 0 \end{aligned}$$

   por lo que se tiene

$$\begin{aligned} \overline{sgn}(0) &= suc(0) \\ \overline{sgn}(suc(y)) &= 0 \end{aligned}$$

   Por tanto, $\overline{sgn}(y) = Recu((suc \circ 0), 0)(y)$, de lo cual se sigue directamente que $\overline{sgn} \in \mathcal{FRP}$ pues se obtiene por el esquema de recursión de funciones en $\mathcal{FRP}$.

7. *Antecesor.* Sea $ant : x \mapsto \begin{cases} 0 & \text{si } x = 0, \\ x - 1 & \text{si } x > 0. \end{cases}$.

   Tenemos

$$\begin{aligned} ant(0) &= 0 \\ ant(y+1) &= y \end{aligned}$$

   por lo que se tiene

$$\begin{aligned} ant(0) &= 0 \\ ant(suc(y)) &= y \end{aligned}$$

   Por tanto, $ant(y) = Recu(0, \pi^1_1)(y)$, de lo cual se sigue directamente que $ant \in \mathcal{FRP}$ pues se obtiene por el esquema de recursión de funciones en $\mathcal{FRP}$.

8. *Diferencia acotada.* Consideremos la función $DA : (x, y) \mapsto x \dot{-} y = \begin{cases} x - y & \text{si } x \geq y, \\ 0 & \text{en otro caso.} \end{cases}$.

   Ya que para cualesquiera $x, y \in \mathbb{N}$ se cumple

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

   se tiene que

$$\begin{aligned} DA(x, 0) &= identidad(x) \\ DA(x, suc(y)) &= ant(DA(x, y)) \end{aligned}$$

   Por tanto, $DA(x, y) = Recu(\pi^1_1, h)(x, y)$, donde $\forall x, y, z : h(x, y, z) = ant \circ \pi^3_3(x, y, z)$, de lo cual se sigue directamente que $DA \in \mathcal{FRP}$ pues se obtiene por el esquema de recursión de funciones en $\mathcal{FRP}$.

9. *Distancia.* Sea $dist : (x, y) \mapsto |x - y| = \begin{cases} x - y & \text{si } x \geq y, \\ y - x & \text{si } x \leq y. \end{cases}$.

   Ya que $\forall x, y : \ dist(x, y) = (x \dot{-} y) + (y \dot{-} x)$ se tiene que $dist \in \mathcal{FRP}$.

Sea $A \subset \mathbb{N}^n$ un subconjunto. La *función característica* de A es la función $\chi_A : \mathbf{x} \mapsto \begin{cases} 1 & \text{si } \mathbf{x} \in A, \\ 0 & \text{si } \mathbf{x} \notin A. \end{cases}$. Así pues, χ_A hace las veces de un oráculo que para cada $\mathbf{x} \in \mathbb{N}^n$ decide si acaso $\mathbf{x} \in A$.

El conjunto $A \subset \mathbb{N}^n$ se dice ser *recursivo primitivo*, y abusando de la notación escribiremos $A \in \mathcal{FRP}$, si su función característica lo es, es decir, si $\chi_A \in \mathcal{FRP}$.

Una relación de aridad n se dice ser *recursiva primitiva*, si ella, vista como un subconjunto de $\mathbb{N}^n$, lo es.

Veamos algunos ejemplos de conjuntos en $\mathcal{FRP}$.

9. "*Mayor que*". Para cualesquiera dos $x, y \in \mathbb{N}$ se tiene

$$x > y \quad \Leftrightarrow \quad sgn(x \dot{-} y) = 1,$$

por tanto $\chi_{(>)} = sgn \circ DA$ de donde se sigue que la relación ">" es, en efecto, recursiva primitiva.

10. *Igualdad*. Para cualesquiera dos $x, y \in \mathbb{N}$ se tiene

$$x = y \quad \Leftrightarrow \quad |x - y| = 0,$$

por tanto $\chi_{(=)} = \overline{sgn} \circ dist$ de donde se sigue que la relación "=" es, en efecto, recursiva primitiva.

Denotemos por $\mathcal{FRP}_n$ a la clase de funciones recursivas primitivas cuyo dominio es $\mathbb{N}^n$.

Proposición 2.1.2 *La clase de conjuntos recursivos primitivos en $\mathbb{N}^n$, que continuando con el abuso de la notación denotaremos como $\mathcal{FRP}_n$, forma un álgebra de conjuntos, es decir,*

$$\begin{aligned} & \emptyset \in \mathcal{FRP}_n \\ A \in \mathcal{FRP}_n \quad &\Rightarrow \quad A^c = (\mathbb{N}^n - A) \in \mathcal{FRP}_n \\ A, B \in \mathcal{FRP}_n \quad &\Rightarrow \quad A \cap B \in \mathcal{FRP}_n \text{ y } A \cup B \in \mathcal{FRP}_n. \end{aligned}$$

Dem. Basta considerar las siguientes identidades:

$$\begin{aligned} \chi_{\emptyset} &= Cero \circ \pi_1^n, \\ \chi_{A^c} &= \left(1 \dot{-} \chi_A\right), \\ \chi_{A \cap B} &= \chi_A \cdot \chi_B, \\ \chi_{A \cup B} &= \left(1 \dot{-} \chi_{(A^c \cap B^c)}\right). \end{aligned}$$

□

Si $A \in \mathbb{N}^n$ es un conjunto y $z \in \mathbb{N}$, la *proyección de A acotada por z*, en sus $(n-1)$ primeras coordenadas, es el conjunto

$$\exists_{\leq z} A = \{\mathbf{x} \in \mathbb{N}^{n-1} | \exists y \leq z : (\mathbf{x}, y) \in A\}.$$

Proposición 2.1.3 *Si $A \in \mathcal{FRP}_n$ entonces para cualquier $z \in \mathbb{N}$ se tiene que $\exists_{\leq z} A \in \mathcal{FRP}_{n-1}$.*

Dem. Esto es evidente de la relación válida $\forall \mathbf{x} \in \mathbb{N}^{n-1}$:

$$\mathbf{x} \in \exists_{\leq z} A \quad \Leftrightarrow \quad sgn\left(\sum_{y=0}^{z} \chi_A(\mathbf{x}, y)\right) = 1.$$

□

Podemos ver ahora esquemas de composición bajo los cuales es cerrada la clase $\mathcal{FRP}$.

11. *Funciones definidas por dos condiciones.* Si $f_1, f_0 \in \mathcal{FRP}$ y $A \in \mathcal{FRP}$ entonces la función
$$f : \mathbf{x} \mapsto \begin{cases} f_0(\mathbf{x}) & \text{si } \mathbf{x} \notin A, \\ f_1(\mathbf{x}) & \text{si } \mathbf{x} \in A. \end{cases}$$
es recursiva primitiva.
En efecto, se tiene que $\forall \mathbf{x}$: $f(\mathbf{x}) = f_1(\mathbf{x}) \cdot \chi_A(\mathbf{x}) + f_0(\mathbf{x}) \cdot \big(1 \dot{-} \chi_A(\mathbf{x})\big)$.

12. *Minimización acotada.* La minimización acotada de cualquier función recursiva primitiva es, a su vez, recursiva primitiva. En otras palabras, la clase $\mathcal{FRP}$ es cerrada bajo el esquema de composisción *MiAc*,
$$f \in \mathcal{FRP} \quad \Rightarrow \quad MiAc(f) \in \mathcal{FRP}.$$
En efecto, para todo $\mathbf{x}$ tenemos que para cualquier y el valor de $MiAc(f)(\mathbf{x}, y)$ coincidirá con el valor y, si no se ha encontrado un valor que anule a $y_1 \mapsto f(\mathbf{x}, y_1)$ con $0 \leq y_1 \leq y$, o bien coincidirá con el valor en el antecesor, $y - 1$, en otro caso. Así pues, tendremos
$$\begin{array}{lcl} MiAc(f)(\mathbf{x}, y) & = & 0 \\ MiAc(f)(\mathbf{x}, y+1) & = & \begin{cases} MiAc(f)(\mathbf{x}, y) & \text{si } \exists_{\leq y+1}\{(\mathbf{x}, y) | f(\mathbf{x}, y) = 0\}, \\ y + 1 & \text{en otro caso.} \end{cases} \end{array}$$
Como todas las funciones y todas las relaciones involucradas son recursivas primitivas tenemos que $MiAc(f) \in \mathcal{FRP}$.

Continuando con nuestra lista de funciones y de relaciones recursivas primitivas, presentamos a las siguientes:

13. *Divisibilidad.* Escribamos $x|y$ para indicar que existe $z \leq x$ tal que $x = zy$, y, en tal caso, se dice que x es *divisible* entre y. Tenemos pues,
$$x|y \quad \Leftrightarrow \quad \exists z \leq x : x = zy.$$
Consecuentemente, "|" es una relación recursiva primitiva.

14. *División.* Sea $\texttt{div} : (x, y) \mapsto \begin{cases} \lfloor \frac{x}{y} \rfloor & \text{si } y \neq 0, \\ 0 & \text{si } y = 0. \end{cases}$.
Si consideramos la función $f : (x, y, z) \mapsto x \dot{-} yz$, la cual es recursiva primitiva, tenemos que $\texttt{div}(x, y) = MiAc(f)(x, y)$ siempre que $y \neq 0$. Se sigue que `div` es recursiva primitiva.

15. *Módulo.* $\texttt{mod} : (x, y) \mapsto x \dot{-} y \cdot \texttt{div}(x, y)$ es recursiva primitiva.

16. *Primos.* Un número es *compuesto*, $x \in Primos^c$, si $\exists y \leq x : (y \neq 1) \wedge (y \neq x) \wedge (x|y)$.
Por tanto la propiedad de ser compuesto es recursiva primitiva.
La propiedad de ser primo es también recursiva primitiva, como complemento de ésa.

17. *Lista de primos.* Según vimos en la demostración del teorema de Euclides acerca de la infinidad de los números primos, el $(n+1)$-ésimo primo p_{n+1} excede a p_n pero no excede al número $q_n = \prod_{i=0}^{n} p_i + 1$.
Así pues, la lista de primos se puede construir con el esquema siguiente:
$$\begin{array}{lcl} p_0 & = & 2 \\ p_{n+1} & = & \text{Min}\{y | \, (p_n < y \leq \prod_{i=0}^{n} p_i + 1) \wedge (y \in Primos)\} \end{array}$$
por tanto la lista de primos se construye de manera recursiva primitiva.

De los ejemplos mostrados, el lector podrá ver que muchas funciones de interés en aritmética son recursivas primitivas. Entre éstas, sin entrar en demostraciones, tan solo mencionamos a los siguientes: el sacar el exponente de un primo en la descomposición en primos de un entero, el sacar la lista de exponentes no-nulos en la descomposición en primos de un entero, el decidir si dos enteros son primos relativos, la función de Euler (que cuenta el número de primos relativos de un número dado menores que él), el símbolo de Legendre, el cual decide para dos primos relativos x, m si acaso x es un *residuo cuadrático* de m (es decir, $\exists x : x^2 \equiv a \bmod m$), etc.

Dos esquemas de reiteración

1. Sea $g : \mathbb{N} \to \mathbb{N}$ una función recursiva primitiva. Definimos a la función $f : \mathbb{N}^2 \to \mathbb{N}$ recursivamente como sigue

$$\begin{aligned} f(0,x) &= g(x) \\ \forall n \geq 0 : \ f(n+1,x) &= f(n, f(n,x)) \end{aligned}$$

   Mostremos que f es una función recursiva primitiva.

   Reescritura de f: Se puede ver que $\forall n \geq 0 : \ f(n,x) = g^{2^n}(x)$. Por tanto, se sigue que

$$\boxed{f \text{ es recursiva primitiva.}}$$

2. Si $g : \mathbb{N} \to \mathbb{N}$ una función recursiva primitiva entonces

$$\boxed{\text{la } \textit{iteración } h : (n,x) \mapsto g^n(x) \text{ es una función recursiva primitiva.}}$$

   En efecto, f es una función recursiva primitiva porque es igual a la composición de una función recursiva primitiva con la función recursiva primitiva anterior.

Ejemplos: Como ejemplos de la segunda función de reiteración arriba descrita la aplicaremos sobre algunas funciones polinomiales.

- *Sucesor.* Supongamos $g(x) = x + 1$. Entonces $f(n,x) = x + 2^n$.
- *Una lineal.* Supongamos $g(x) = 2x + 1$. Entonces $f(n,x) = 2^{2^n}x + (2^{2^n} - 1)$.
- *Caso lineal general.* Supongamos $g(x) = ax + b$. Entonces

$$f(n,x) = a^{2^n}x + b\left(\frac{a^{2^n} - 1}{a - 1}\right).$$

- *Caso cuadrático general.* Supongamos $g(x) = ax^2 + bx + c$. Entonces

$$\begin{aligned} f(0,x) &= c + bx + ax^2 \\ f(1,x) &= \left(c + bc + ac^2\right) + \\ &\quad \left(b^2 + 2abc\right)x + \\ &\quad \left(ab + ab^2 + 2a^2c\right)x^2 + \\ &\quad \left(2a^2b\right)x^3 + \\ &\quad a^3x^4 \end{aligned}$$

   Para el siguiente "renglón", los valores de $f(2,x)$ se muestran en la tabla 2.1.

2.2 Funciones elementales

2.2.1 Algunos otros esquemas de composición

Esquemas de sumatoria y productos acotados

Presentamos de manera general el ejemplo 4. en la lista de funciones recursivas primitivas.

$$
\begin{aligned}
f(2,x) \quad = \quad & \left(\begin{array}{l} c + bc + b^2c + b^3c + ac^2 + 3abc^2 + \\ 6ab^2c^2 + 3ab^3c^2 + ab^4c^2 + 2a^2c^3 + 10a^2bc^3 + 12a^2b^2c^3 + \\ 8a^2b^3c^3 + 2a^2b^4c^3 + 5a^3c^4 + 15a^3bc^4 + 19a^3b^2c^4 + 10a^3b^3c^4 + \\ a^3b^4c^4 + 6a^4c^5 + 18a^4bc^5 + 18a^4b^2c^5 + 4a^4b^3c^5 + 6a^5c^6 + \\ 14a^5bc^6 + 6a^5b^2c^6 + 4a^6c^7 + 4a^6bc^7 + a^7c^8 \end{array}\right) + \\
& \left(\begin{array}{l} b^4 + 6ab^3c + 4ab^4c + 2ab^5c + 12a^2b^2c^2 + 20a^2b^3c^2 + \\ 18a^2b^4c^2 + 6a^2b^5c^2 + 8a^3bc^3 + 32a^3b^2c^3 + 52a^3b^3c^3 + 36a^3b^4c^3 + \\ 4a^3b^5c^3 + 16a^4bc^4 + 60a^4b^2c^4 + 78a^4b^3c^4 + 20a^4b^4c^4 + 24a^5bc^5 + \\ 72a^5b^2c^5 + 36a^5b^3c^5 + 24a^6bc^6 + 28a^6b^2c^6 + 8a^7bc^7 \end{array}\right) x + \\
& \left(\begin{array}{l} ab^3 + ab^4 + ab^5 + ab^6 + 6a^2b^2c + 8a^2b^3c + \\ 12a^2b^4c + 12a^2b^5c + 6a^2b^6c + 12a^3bc^2 + 24a^3b^2c^2 + 42a^3b^3c^2 + \\ 54a^3b^4c^2 + 48a^3b^5c^2 + 6a^3b^6c^2 + 8a^4c^3 + 32a^4bc^3 + 68a^4b^2c^3 + \\ 108a^4b^3c^3 + 136a^4b^4c^3 + 40a^4b^5c^3 + 16a^5c^4 + 60a^5bc^4 + 114a^5b^2c^4 + \\ 170a^5b^3c^4 + 90a^5b^4c^4 + 24a^6c^5 + 72a^6bc^5 + 96a^6b^2c^5 + 84a^6b^3c^5 + \\ 24a^7c^6 + 28a^7bc^6 + 28a^7b^2c^6 + 8a^8c^7 \end{array}\right) x^2 + \\
& \left(\begin{array}{l} 2a^2b^3 + 2a^2b^4 + 4a^2b^5 + 2a^2b^6 + 2a^2b^7 + 8a^3b^2c + \\ 20a^3b^3c + 32a^3b^4c + 28a^3b^5c + 28a^3b^6c + 4a^3b^7c + 8a^4bc^2 + \\ 48a^4b^2c^2 + 104a^4b^3c^2 + 132a^4b^4c^2 + 120a^4b^5c^2 + 40a^4b^6c^2 + 32a^5bc^3 + \\ 144a^5b^2c^3 + 288a^5b^3c^3 + 240a^5b^4c^3 + 120a^5b^5c^3 + 72a^6bc^4 + 300a^6b^2c^4 + \\ 260a^6b^3c^4 + 140a^6b^4c^4 + 120a^7bc^5 + 168a^7b^2c^5 + 56a^7b^3c^5 + 56a^8bc^6 \end{array}\right) x^3 + \\
& \left(\begin{array}{l} a^3b^2 + a^3b^3 + 7a^3b^4 + 7a^3b^5 + 7a^3b^6 + 6a^3b^7 + \\ a^3b^8 + 4a^4bc + 10a^4b^2c + 36a^4b^3c + 56a^4b^4c + 90a^4b^5c + \\ 54a^4b^6c + 20a^4b^7c + 4a^5c^2 + 24a^5bc^2 + 72a^5b^2c^2 + 156a^5b^3c^2 + \\ 336a^5b^4c^2 + 210a^5b^5c^2 + 90a^5b^6c^2 + 16a^6c^3 + 72a^6bc^3 + 204a^6b^2c^3 + \\ 520a^6b^3c^3 + 420a^6b^4c^3 + 140a^6b^5c^3 + 36a^7c^4 + 150a^7bc^4 + 330a^7b^2c^4 + \\ 420a^7b^3c^4 + 70a^7b^4c^4 + 60a^8c^5 + 84a^8bc^5 + 168a^8b^2c^5 + 28a^9c^6 \end{array}\right) x^4 + \\
& \left(\begin{array}{l} 6a^4b^3 + 10a^4b^4 + 10a^4b^5 + 24a^4b^6 + 10a^4b^7 + 4a^4b^8 + \\ 24a^5b^2c + 56a^5b^3c + 108a^5b^4c + 180a^5b^5c + 104a^5b^6c + 36a^5b^7c + \\ 24a^6bc^2 + 108a^6b^2c^2 + 348a^6b^3c^2 + 480a^6b^4c^2 + 384a^6b^5c^2 + 84a^6b^6c^2 + \\ 72a^7bc^3 + 480a^7b^2c^3 + 600a^7b^3c^3 + 560a^7b^4c^3 + 56a^7b^5c^3 + 240a^8bc^4 + \\ 420a^8b^2c^4 + 280a^8b^3c^4 + 168a^9bc^5 \end{array}\right) x^5 + \\
& \left(\begin{array}{l} 2a^5b^2 + 8a^5b^3 + 8a^5b^4 + 36a^5b^5 + 36a^5b^6 + 22a^5b^7 + \\ 6a^5b^8 + 8a^6bc + 28a^6b^2c + 64a^6b^3c + 256a^6b^4c + 240a^6b^5c + \\ 184a^6b^6c + 28a^6b^7c + 8a^7c^2 + 36a^7bc^2 + 144a^7b^2c^2 + 580a^7b^3c^2 + \\ 660a^7b^4c^2 + 420a^7b^5c^2 + 28a^7b^6c^2 + 24a^8c^3 + 160a^8bc^3 + 480a^8b^2c^3 + \\ 840a^8b^3c^3 + 280a^8b^4c^3 + 80a^9c^4 + 140a^9bc^4 + 420a^9b^2c^4 + 56a^{10}c^5 \end{array}\right) x^6 + \\
& \left(\begin{array}{l} 4a^6b^2 + 4a^6b^3 + 24a^6b^4 + 64a^6b^5 + 52a^6b^6 + 36a^6b^7 + \\ 4a^6b^8 + 8a^7bc + 24a^7b^2c + 168a^7b^3c + 320a^7b^4c + 384a^7b^5c + \\ 168a^7b^6c + 8a^7b^7c + 24a^8bc^2 + 360a^8b^2c^2 + 600a^8b^3c^2 + 840a^8b^4c^2 + \\ 168a^8b^5c^2 + 240a^9bc^3 + 560a^9b^2c^3 + 560a^9b^3c^3 + 280a^{10}bc^4 \end{array}\right) x^7 + \\
& \left(\begin{array}{l} a^7b + a^7b^2 + 6a^7b^3 + 61a^7b^4 + 70a^7b^5 + 90a^7b^6 + \\ 28a^7b^7 + a^7b^8 + 2a^8c + 6a^8bc + 42a^8b^2c + 260a^8b^3c + \\ 420a^8b^4c + 420a^8b^5c + 56a^8b^6c + 6a^9c^2 + 90a^9bc^2 + 330a^9b^2c^2 + \\ 840a^9b^3c^2 + 420a^9b^4c^2 + 60a^{10}c^3 + 140a^{10}bc^3 + 560a^{10}b^2c^3 + 70a^{11}c^4 \end{array}\right) x^8 + \\
& \left(\begin{array}{l} 30a^8b^3 + 60a^8b^4 + 120a^8b^5 + 84a^8b^6 + 8a^8b^7 + 120a^9b^2c + \\ 260a^9b^3c + 560a^9b^4c + 168a^9b^5c + 120a^{10}bc^2 + 420a^{10}b^2c^2 + 560a^{10}b^3c^2 + \\ 280a^{11}bc^3 \end{array}\right) x^9 + \\
& \left(\begin{array}{l} 6a^9b^2 + 34a^9b^3 + 90a^9b^4 + 140a^9b^5 + 28a^9b^6 + 24a^{10}bc + \\ 96a^{10}b^2c + 420a^{10}b^3c + 280a^{10}b^4c + 24a^{11}c^2 + 84a^{11}bc^2 + 420a^{11}b^2c^2 + \\ 56a^{12}c^3 \end{array}\right) x^{10} + \\
& \left(\begin{array}{l} 12a^{10}b^2 + 36a^{10}b^3 + 140a^{10}b^4 + 56a^{10}b^5 + 24a^{11}bc + 168a^{11}b^2c + \\ 280a^{11}b^3c + 168a^{12}bc^2 \end{array}\right) x^{11} + \\
& \left(\begin{array}{l} 2a^{11}b + 6a^{11}b^2 + 84a^{11}b^3 + 70a^{11}b^4 + 4a^{12}c + 28a^{12}bc + \\ 168a^{12}b^2c + 28a^{13}c^2 \end{array}\right) x^{12} + \\
& \left(\begin{array}{l} 28a^{12}b^2 + 56a^{12}b^3 + 56a^{13}bc \end{array}\right) x^{13} + \\
& \left(\begin{array}{l} 4a^{13}b + 28a^{13}b^2 + 8a^{14}c \end{array}\right) x^{14} + \\
& 8a^{14}bx^{15} + \\
& a^{15}x^{16}
\end{aligned}
$$

Tabla 2.1: Valores de $f(2,x)$.

$$\boxed{Suma, Prod : (\mathbb{N})^{\mathbb{N}^n} \to (\mathbb{N})^{\mathbb{N}^n}.}$$

Si $g : \mathbb{N}^n \to \mathbb{N}$ es una función definimos

$$\begin{array}{rcl} Suma_1 : \mathbb{N}^n & \to & \mathbb{N} \\ (m, \mathbf{x}) & \mapsto & Suma_1(m, \mathbf{x}) = \sum_{j=0}^{m} g(j, \mathbf{x}) \end{array} \quad ; \quad \begin{array}{rcl} Prod_1 : \mathbb{N}^n & \to & \mathbb{N} \\ (m, \mathbf{x}) & \mapsto & Prod_1(m, \mathbf{x}) = \prod_{j=0}^{m} g(j, \mathbf{x}) \end{array}$$

Esquema de transformación explícita

En este esquema, dada una función $g : \mathbb{N}^n \to \mathbb{N}$, podemos obtener varias funciones, simplemente instanciando algunas variables que aparezcan como argumentos por constantes, o bien por algunas otras variables. Cada posible "instanciación" da lugar a una nueva función. Tenemos,

$$\boxed{TrEx : (\mathbb{N})^{\mathbb{N}^n} \to \mathcal{P}\left(\bigcup_{j=0}^{n} (\mathbb{N})^{\mathbb{N}^j}\right).}$$

Más precisamente:

Si $g : \mathbb{N}^n \to \mathbb{N}$ es una función, para $j \leq n$ decimos que una función $f^j : \mathbb{N}^j \to \mathbb{N}$ se obtiene por una *tranformación explícita* a partir de g si es de la forma

$$\mathbf{x} \mapsto f^j(\mathbf{x}) = g\left(\xi_1(\mathbf{x}), \ldots, \xi_j(\mathbf{x}), \ldots, \xi_n(\mathbf{x})\right)$$

donde cada ξ_k es bien una proyección o una función constante.

2.2.2 Clase de funciones elementales

Esta clase de funciones es bastante amplia y suficiente para el cálculo de la mayoría de las aplicaciones en ingeniería.

En la tabla 2.2 presentamos varias clases de funciones. Probaremos que todas ellas coinciden entre sí. Las funciones en cada una de esas clases son llamadas *elementales*. Todos los polinomios, por ejemplo, son funciones elementales. Se sigue de inmediato la siguiente,

Proposición 2.2.1 *La clase $\mathcal{E}$ de las funciones elementales está incluída en la clase $\mathcal{FRP}$ de las funciones recursivas primitivas.*

Teorema 2.2.1 *Las clases $\mathcal{F}^1$, $\mathcal{F}^2$ y $\mathcal{F}^3$ coinciden con la clase $\mathcal{E}$ de funciones elementales.*

Hay que probar las inclusiones $\mathcal{E} \subset \mathcal{F}^1 \subset \mathcal{F}^2 \subset \mathcal{F}^3 \subset \mathcal{E}$.

En nuestra presentación no abundaremos en detalles. Presentaremos más bien meras indicaciones de cómo desarrollar esas demostraciones.

Para demostrar una inclusión de la forma $\mathcal{F} \subset \mathcal{G}$, donde $\mathcal{F}$ y $\mathcal{G}$ son dos clases de funciones definidas recursivamente, hay que ver que

- las funciones básicas de $\mathcal{F}$ están en $\mathcal{G}$, y
- $\mathcal{G}$ es cerrado bajo los esquemas de composición de $\mathcal{F}$, o lo que es lo mismo, que los esquemas de $\mathcal{F}$ se construyen con los de $\mathcal{G}$.

$\mathcal{E}$ es la clase mínima de funciones tal que

- contiene a las funciones
  - sucesor: $suc : x \mapsto x + 1$,
  - suma: $+ : (x, y) \mapsto x + y$,
  - diferencia acotada: $\dot{-} : (x, y) \mapsto x \dot{-} y$
- y es cerrada bajo los esquemas de
  - composición,
  - transformación explícita, y
  - sumatoria y productos acotados.

$\mathcal{F}^1$ es la clase mínima de funciones tal que

- contiene a las funciones
  - sucesor: $suc : x \mapsto x + 1$,
  - diferencia acotada: $\dot{-} : (x, y) \mapsto x \dot{-} y$
  - exponenciación: $\cdot^{\cdot} : (x, y) \mapsto x^y$
- y es cerrada bajo los esquemas de
  - composición,
  - transformación explícita, y
  - minimización acotada.

$\mathcal{F}^2$ es la clase mínima de funciones tal que

- contiene a las funciones
  - sucesor: $suc : x \mapsto x + 1$,
  - exponenciación: $\cdot^{\cdot} : (x, y) \mapsto x^y$
- y es cerrada bajo los esquemas de
  - composición,
  - transformación explícita, y
  - recursión acotada.

$\mathcal{F}^3$ es la clase mínima de funciones tal que

- contiene a las funciones
  - sucesor: $suc : x \mapsto x + 1$,
  - diferencia acotada: $\dot{-} : (x, y) \mapsto x \dot{-} y$
  - producto: $\cdot : (x, y) \mapsto xy$,
  - exponenciación: $\cdot^{\cdot} : (x, y) \mapsto x^y$
- y es cerrada bajo los esquemas de
  - composición,
  - transformación explícita, y
  - sumatoria acotada.

Tabla 2.2: Definiciones alternativas de las funciones elementales.

Demostración del teorema. Mostremos que se cumple cada una de las inclusiones necesarias.

1. $\mathcal{E} \subset \mathcal{F}^1$:

   (a) En cuanto a las funciones básicas de $\mathcal{E}$, tenemos que la función sucesor y la diferencia acotada están en $\mathcal{F}^1$ *ex definitione*. Para ver que la suma está en $\mathcal{F}^1$ probemos de manera más general, la proposición siguiente:

   (Producto y suma con exponenciación y MiAc): La suma y el producto usuales de los número naturales se pueden construir mediante la función exponenciación y el esquema de minimización acotada.

   En efecto, dado que

   $$\begin{aligned} a^{b+c} &= a^b a^c \\ a^{bc} &= \left(a^b\right)^c \end{aligned}$$

   tenemos

   $$\begin{aligned} x \cdot y &= \text{Min}\{z \leq (x+1)^{y+1} | 2^z = (2^x)^y\} \\ x + y &= \text{Min}\{z \leq (x+1)\cdot(y+1) | 2^z = (2^x)(2^y)\} \end{aligned}$$

   (b) Observemos ahora algunas propiedades de $\mathcal{F}^1$:

      i. *Los conjuntos en $\mathcal{F}^1$ forman un álgebra de conjuntos*: es decir, $\chi_\emptyset \in \mathcal{F}^1$ y si A, B son dos conjuntos tales que $\chi_A, \chi_B \in \mathcal{F}^1$ entonces $\chi_{A^c}, \chi_{A\cap B} \in \mathcal{F}^1$.
      En efecto, la función cero, que es $\chi_\emptyset$, está en $\mathcal{F}^1$ pues esta clase es cerrada por transformación explícita.
      La otra aseveración se sigue de las relaciones

      $$\begin{aligned} x = 0 &\Leftrightarrow 0^x = 1 \\ (x = 0) \wedge (y = 0) &\Leftrightarrow x^{0^y} = 0 \end{aligned}$$

      ii. *$\mathcal{F}^1$ es cerrada bajo proyecciones acotadas.*
      Esto se prueba de manera similar a como se vió que las funciones recursivas primitivas son cerradas bajo proyecciones acotadas.

      iii. *$\mathcal{F}^1$ es cerrada bajo el esquema de recursión acotada.*
      Debido a que es cerrada por proyecciones acotadas, puede verse que la relación de ser primo y el tomar descomposiciones en primos de un entero dado son procedimientos en $\mathcal{F}^1$.
      Por tanto, podemos codificar secuencias por números en $\mathcal{F}^1$:

      $$\mathbf{m} = (m_0, \ldots, m_k) \in \mathbb{N}^* \leftrightarrow \lceil \mathbf{m} \rceil = \prod_{i=0}^{k} p_i^{m_i}.$$

      Con esto se puede ver que, en efecto, $\mathcal{F}^1$ es cerrada bajo el esquema de recursión acotada.

   (c) En cuanto a las reglas de composición, basta ver que $\mathcal{F}^1$ es cerrada bajo la sumatoria y el producto acotados.

   Pero esto es una consecuencia de que $\mathcal{F}^1$ es cerrada bajo la recursión acotada.

2. $\mathcal{F}^1 \subset \mathcal{F}^2$:

   (a) En cuanto a las funciones básicas de $\mathcal{F}^1$, tenemos que la función sucesor y la exponenciación están en $\mathcal{F}^2$ *ex definitione*. La función diferencia acotada, así como la suma y el producto, en $\mathcal{F}^2$ debido a las *leyes de exponenciación*: $a^{x+y} = a^x a^y$, $a^{xy} = (a^x)^y$.

   (b) $\mathcal{F}^2$ es cerrado bajo el esquema de minimización acotada debido a que este esquema es constructible en $\mathcal{F}^2$.

3. $\mathcal{F}^2 \subset \mathcal{F}^3$: Para ver esto, basta mostrar que $\mathcal{F}^3$ es cerrado bajo el esquema de recursión acotada. Se debe, entonces, demostrar que $\mathcal{F}^3$ es cerrado bajo el esquema de minimización acotada.

   Sea $f \in \mathcal{F}^3$. Consideremos la función $f_1 : (\mathbf{x}, y) \mapsto \sum_{z=0}^{y} \left[1 \dot{-} f(\mathbf{x}, z)\right]$, que está en $\mathcal{F}^3$. Tenemos que se cumplen las equivalencias lógicas siguientes:

$$\begin{aligned} x \neq 0 &\Leftrightarrow 1 \dot{-} x = 0, \\ x = 0 &\Leftrightarrow 1 \dot{-} x = 1, \\ f_1(\mathbf{x}, y) = 0 &\Leftrightarrow \forall z \leq y : f(\mathbf{x}, y) \neq 0, \\ \sum_{z=0}^{y} \left[1 \dot{-} f(\mathbf{x}, z)\right] = a &\Leftrightarrow \forall z : [0 \leq z < a \Rightarrow f(\mathbf{x}, z) \neq 0]. \end{aligned}$$

   Sea pues $g : (\mathbf{x}, y) \mapsto \left(\sum_{z=0}^{y} \left[1 \dot{-} f_1(\mathbf{x}, z)\right]\right) \cdot \left(1 \dot{-} \left(\left(\sum_{z=0}^{y} \left[1 \dot{-} f_1(\mathbf{x}, z)\right]\right) \dot{-} y\right)\right)$. Es evidente que $g \in \mathcal{F}^3$ y además puede verse que $g = MiAc(f)$.

   Ya que $\mathcal{F}^3$ es cerrada por minimización acotada, lo es también por recursión acotada.

4. $\mathcal{F}^3 \subset \mathcal{E}$: Esta inclusión rige *ex definitione* también. □

2.3 Jerarquía de Grzegorczyk

Veremos en esta sección que la clase de las funciones recursivas primitivas contiene una sucesión de funciones tal que cada función en ella tiene un crecimiento sustantivamente más complejo que la función que la precede en la sucesión. Tal escala de funciones nos permitirá definir a la *jerarquía de Grzegorczyk* en $\mathcal{FRP}$, con la cual podremos clasificar a las funciones recursivas primitivas en diversos "niveles de complejidad".

2.3.1 Una escala de funciones

Consideremos la sucesión $F = \{f_n : \mathbb{N}^2 \to \mathbb{N}\}_{n \in \mathbb{N}}$ definida como sigue:

$$f_0(x, y) = y + 1 \tag{2.1}$$
$$f_1(x, y) = x + y \tag{2.2}$$
$$f_2(x, y) = (x + 1)(y + 1) \tag{2.3}$$

y para cada $n \geq 3$ definimos inductivamente

$$f_n(0, y) = f_{n-1}(y + 1, y + 1) \tag{2.4}$$
$$\forall x > 0 : \; f_n(x, y) = f_n(x - 1, f_n(x - 1, y)) \tag{2.5}$$

La relación (2.4) indica que, en "su primer renglón", la n-ésima función asume los valores en la diagonal de la $(n-1)$-ésima, corridos una posición hacia adelante.

Ahora bien, hablando de manera tosca y ambigua en demasía, podríamos decir que la relación (2.5) indica que, en "su x-ésimo renglón", la n-ésima función está "reiterando x veces los valores previos de esa misma n-ésima función".

La primera función construída según (2.4) y (2.5), a saber, f_3, tiene, en cada renglón, un crecimiento polinomial muy rápido en términos de y, pero, en cada columna, tiene un crecimiento doblemente exponencial en términos de x. En la tabla 2.3 presentamos las formas que asume $f_3(x, y)$ haciendo sustituciones para los primeros valores de x.

$$
\begin{aligned}
f_3(0,y) &= (2+y)^2 \\
&= 4+4y+y^2 \\
\\
f_3(1,y) &= \left(2+(2+y)^2\right)^2 \\
&= 36+48y+28y^2+8y^3+y^4 \\
\\
f_3(2,y) &= (2+(2+(2+(2+y)^2)^2)^2)^2 \\
&= 2090916+10550016y+26125248y^2+41867904y^3+ \\
&\quad 48398416y^4+42666880y^5+29610272y^6+16475584y^7+ \\
&\quad 7419740y^8+2711424y^9+800992y^{10}+189248y^{11}+ \\
&\quad 35064y^{12}+4928y^{13}+496y^{14}+32y^{15}+y^{16} \\
\\
f_3(3,y) &= \left(2+\left(2+\left(2+\left(2+\left(2+\left(2+\left(2+(2+y)^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2
\end{aligned}
$$

Tabla 2.3: Formas de $f_3(x,y)$, con $x = 0,1,2,3$.

De hecho, puede verse que, en general,

$$f_3(x,y) = \underbrace{(2+(2+\ldots(2+}_{2^x \text{ veces}} y \underbrace{)^2\ldots)^2)^2}_{2^x \text{ veces}}.$$

así pues $f_3(x,y)$ es un polinomio en y de grado 2^{2^x}.

Por tanto, tendremos que $f_4(0,\cdot) : y \mapsto \underbrace{(2+(2+\ldots(2+}_{2^{(y+1)} \text{ veces}} (y+1) \underbrace{)^2\ldots)^2)^2}_{2^{(y+1)} \text{ veces}}$.

Aquí ya es evidente que se carece de notación "aritmética" para escribir a $f_4(1,\cdot) : y \mapsto f_4(0, f_4(0,y))$, ya sin decir más de $f_4(x,\cdot)$ con $x \geq 2$.

$f_5(0,\cdot)$ toma los valores en la diagonal de f_4 y reinicia el proceso.

Et cetera.

No es presuntuoso afirmar que el crecimiento de f_{10} o de f_{1998} o de $f_{1998^{1998}}$ supera, con mucho, los alcances de nuestro propio entendimiento.

2.3.2 Propiedades de la escala

Consideremos la sucesión de funciones $\{g_n : \mathbb{N} \to \mathbb{N}\}_{n\geq 3}$ definida recursivamente como sigue:

$$
\begin{aligned}
g_3 : y &\mapsto (2+y)^2 \\
\forall n \geq 3 \quad g_{n+1} : y &\mapsto g_n^{2^{y+1}}(y+1)
\end{aligned}
\tag{2.6}
$$

(aquí, la notación g^e significa la composición de g consigo misma e veces).

Proposición 2.3.1 $\forall n \geq 3\ \forall x,y \geq 0 :\ f_n(x,y) = g_n^{2^x}(y)$.

DEM. Demostrémosla por inducción en n.

Caso "base": Para $n = 3$ se tiene

$$g_3^{2^x}(y) = \underbrace{(2 + (2 + \ldots (2+}_{2^x \text{ veces}} y \underbrace{)^2 \ldots)^2)^2}_{2^x \text{ veces}} = f_3(x, y) \tag{2.7}$$

según se vió anteriormente.

Caso inductivo: Supongamos que para $n \geq 3$ se cumple $f_n(x, y) = g_n^{2^x}(y)$.

Ahora, mostremos por inducción en x la igualdad

$$f_{n+1}(x, y) = g_{n+1}^{2^x}(y) \tag{2.8}$$

De (2.4) se tiene $f_{n+1}(0, y) = g_n^{2^{y+1}}(y + 1)$. Y de (2.6) se sigue que, en efecto, $f_{n+1}(0, y) = g_{n+1}(y)$.

Supongamos que para $x \geq 0$ se cumple (2.8). Entonces

$$\begin{array}{rcll} f_{n+1}(x+1, y) & = & f_{n+1}(x, f_{n+1}(x, y)) & : \text{por (2.5)} \\ & = & f_{n+1}(x, g_{n+1}^{2^x}(y)) & : \text{por (2.8)} \\ & = & g_{n+1}^{2^x}\left(g_{n+1}^{2^x}(y)\right) & : \text{por (2.8)} \\ & = & g_{n+1}^{2^{x+1}}(y) & : \text{por la ley de las potencias.} \end{array}$$

quod erat demonstratum (qed) □

Proposición 2.3.2 *Se cumplen las propiedades siguientes:*

1. *Toda* sección *de f_n está por encima de la identidad, i.e.*

$$\forall n > 1,\ \forall x, y : y < f_n(x, y).$$

2. *Cada f_n es creciente en su primera componente, i.e.*

$$\forall n > 0,\ \forall x, y : f_n(x, y) < f_n(x + 1, y).$$

3. *Cada f_n es creciente en su segunda componente, i.e.*

$$\forall n > 0,\ \forall x, y : f_n(x, y) < f_n(x, y + 1).$$

4. *La sucesión de funciones $\{f_n\}_n$ es creciente respecto a su índice n, i.e.*

$$\forall n > 0,\ \forall x, y : f_n(x, y) < f_{n+1}(x, y).$$

DEM. Mostraremos cada una de las aseveraciones en la proposición.

1. Por inducción en n:

   Casos "bases": Como f_0 es la función sucesor de su segundo argumento, tenemos evidentemente, $\forall y :\ y < f_0(x, y)$.

   f_1 es la función suma, luego $\forall x, y :\ y \leq f_1(x, y)$, y la desigualdad es estricta siempre que $x > 0$.

   Ahora, $\forall x$ es evidente que se cumple $\forall y :\ y < y + 1 \leq (x + 1)(y + 1) = f_2(x, y)$.

   Caso inductivo: Sea $n \geq 2$. Supongamos que $\forall x, y :\ y < f_n(x, y)$.

   Para probar $\forall y :\ y < f_{n+1}(x, y)$, procedamos por inducción en x. Basta observar que se cumplen,

$$\begin{array}{ll} \forall y : & y < f_n(y + 1, y + 1) = f_{n+1}(0, y) \\ \forall y : & y < f_{n+1}(x, y) < f_{n+1}(x, f_{n+1}(x, y)) = f_{n+1}(x + 1, y) \end{array}$$

2. Comprobemos la aseveración en cada n.

   f_0 es constante respecto a su primer argumento. Es la única función en la escala que no es estrictamente creciente en su primer argumento.

   Para f_1 es evidente que se cumple $\forall x$:

   $$\forall y: \ f_1(x,y) = x + y < (x+1) + y = f_1(x+1, y).$$

   Para f_2 es también evidente que se cumple $\forall x$:

   $$\forall y: \ f_2(x,y) = (x+1)(y+1) < ((x+1)+1)(y+1) = f_2(x+1, y).$$

   Sea $n \geq 2$. Se tiene,

   $$\begin{array}{rll} & f_n(x,y) < f_n(x, f_n(x,y)) & : \text{por lo demostrado en el inciso anterior,} \\ \Rightarrow & f_n(x,y) < f_n(x+1, y) & : \text{por la relación (2.5)} \end{array}$$

3. Procedamos aquí también por inducción en n.

   Casos "bases": Para $n \leq 2$ se comprueba inmediatamente que $\forall x: \ \ y \mapsto f_n(x,y)$ es creciente.

   Caso inductivo: Sea $n \geq 0$. Supongamos que $\forall x: \ \ y \mapsto f_n(x,y)$ es creciente. Veamos que lo mismo pasa para $n+1$. Probaremos esto último por induccón en x. Se tiene,

   $$\begin{array}{rcll} f_{n+1}(0,y) & = & f_n(y+1, y+1) & : \text{por la relación (2.4),} \\ & < & f_n(y+1, y+2) & : \text{por la hipótesis de inducción,} \\ & < & f_n(y+2, y+2) & : \text{por lo probado en el inciso anterior,} \\ & = & f_{n+1}(0, y+1) & : \text{por la relación (2.4).} \end{array}$$

   Sea $x \geq 0$. Supongamos $\forall y: f_{n+1}(x,y) < f_{n+1}(x, y+1)$. Tenemos,

   $$\begin{array}{rcll} f_{n+1}(x+1,y) & = & f_{n+1}(x, f_{n+1}(x,y)) & : \text{por la relación (2.5),} \\ & < & f_{n+1}(x, f_{n+1}(x, y+1)) & : \text{por ambas hipótesis de inducción,} \\ & = & f_{n+1}(x+1, y+1) & : \text{por la relación (2.5).} \end{array}$$

4. Observemos primeramente que para cualquier n,

   $$x \leq y \quad \Rightarrow \quad f_n(x,y) \leq f_{n+1}(x,y).$$

   En efecto,

   $$\begin{array}{rcll} f_n(x,y) & \leq & f_n(y,y) & : \text{pues } x \leq y, \\ & < & f_n(y+1, y+1) & : \text{porque } f_n \text{ es creciente en ambas componentes,} \\ & = & f_{n+1}(0, y+1) & : \text{por la relación (2.4),} \\ & < & f_{n+1}(x, y+1) & : \text{porque } f_n \text{ es creciente en ambas componentes.} \end{array}$$

   Mostremos ahora por inducción en x,

   $$x \geq y \quad \Rightarrow \quad f_n(x,y) \leq f_{n+1}(x,y).$$

   El caso "base" $x = y$ queda subsumido en el caso anterior.

   Supongamos la desigualdad válida para $x \geq y$, cualquiera que sea y. Mostrémosla para $x+1$:

   $$\begin{array}{rcll} f_{n+1}(x+1,y) & = & f_{n+1}(x, f_{n+1}(x,y)) & : \text{por la relación (2.5),} \\ & > & f_n(x, f_n(x,y)) & : \text{por la hipótesis de inducción,} \\ & = & f_n(x+1, y) & : \text{por la relación (2.5).} \end{array}$$

□

2.3.3 Niveles de la jerarquía

Para cada $n \geq 0$ consideremos la clase de funciones "generada" por la función f_n, es decir, para cada $n \geq 0$,

$\mathcal{E}^n$ es la clase mínima de funciones tal que

- contiene a las funciones
  - sucesor: $suc : x \mapsto x + 1$
  - proyecciones: $U_1 : (x, y) \mapsto x$, $U_2 : (x, y) \mapsto y$.
  - $f_n : (x, y) \mapsto f_n(x, y)$,
- y es cerrada bajo los esquemas de
  - composición,
  - transformación explícita, y
  - recursión acotada.

Proposición 2.3.3 *$\mathcal{E}^3$ coincide con la clase de las funciones elementales, es decir, $\mathcal{E} = \mathcal{E}^3$.*

Dem. Consideremos las presentaciones de las funciones elementales en la tabla 2.2.

Veamos que $\mathcal{F}^3 \subset \mathcal{E}^3$. Para esto basta definir a la exponenciación en $\mathcal{E}^3$.

Puede verse que para cualesquiera x, y se cumple $x^y \leq f_3(x, y)$. Ahora, es claro que la exponenciación puede definirse por el esquema de recursión utilizando al producto usual, y, de hecho, utilizando a f_3 la recursión puede hacerse de manera acotada. Por tanto, la exponenciación está en $\mathcal{E}^3$, $[(x, y) \mapsto x^y] \in \mathcal{E}^3$.

Para probar la inclusión opuesta basta ver que f_3 se puede definir en $\mathcal{F}^3$.

De las igualdades en (2.7), al considerar el polinomio cuadrático $g : y \mapsto (2 + y)^2$ y al hacer

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

vemos que $h(x, y) \leq (2 + y)^{2^{2^x}}$ y, según (2.7), $f_3(x, y) = h(2^x, y)$.

De todo esto resulta que f_3 es efectivamente constructible en $\mathcal{F}^3$. □

Proposición 2.3.4 *Las primeras cuatro clases de la escala están anidadas de manera creciente,*

$$\mathcal{E}^0 \subset \mathcal{E}^1 \subset \mathcal{E}^2 \subset \mathcal{E}^3.$$

Dem. Cada una de las inclusiones enlistadas se demuestra directamente. □

Proposición 2.3.5 *Para todo n: $\mathcal{E}^n \subset \mathcal{E}^{n+1}$.*

Dem. Veamos que rige la implicación

$$i \leq n \;\Rightarrow\; f_i \in \mathcal{E}^n.$$

Vimos que $f_i(x, y) = g_i^{2^{2^x}}(y)$. Comprobaremos que la expresión del lado derecho se construye en $\mathcal{E}^n$. Definamos, como lo hicimos para f_3,

$$\begin{array}{lcl} h_i(0, y) & = & y \\ h_i(x + 1, y) & = & g_i(h_i(x, y)) \end{array}$$

entonces $h_i(x,y) \leq g_i^{2^{2^x}}(y)$ y, según la proposición 2.3.1, $f_i(x,y) = h(2^x, y)$.

De todo esto resulta que f_i es efectivamente constructible en $\mathcal{F}^n$, siempre que $i \leq n$. □

Proposición 2.3.6 *La clase de funciones recursivas primitivas coincide con la unión de las $\mathcal{E}^n$'s, es decir*

$$\mathcal{FRP} = \bigcup_{n\geq 0} \mathcal{E}^n.$$

DEM. $\mathcal{FRP}$ es cerrada bajo los esquemas de formación de cada $\mathcal{E}^n$. Como también las funciones básicas de $\mathcal{E}^n$ son recursivas primitivas se tiene $\forall n : \mathcal{E}^n \subset \mathcal{FRP}$ y consecuentemente $\bigcup_{n\geq 0} \mathcal{E}^n \subset \mathcal{FRP}$.

Para demostrar la inclusión $\mathcal{FRP} \subset \bigcup_{n\geq 0} \mathcal{E}^n$, hay que demostrar, y nosotros omitiremos aquí su demostración, el siguiente

Lema 2.3.1 *Si $f \in \mathcal{FRP}$ requiere de la aplicación de n reglas, sean de composición o de recursión, para ser definida en $\mathcal{FRP}$, entonces necesariamente $f \in \mathcal{E}^{n+3}$.*

□

Ahora, puede verse que $\forall n : \mathcal{E}^{n+1} - \mathcal{E}^n \neq \emptyset$.

Proposición 2.3.7 *La diagonal de f_{n+1} crece más rápido que cualquier función en $\mathcal{E}^n$, es decir*

$$\forall f \in \mathcal{E}^n \exists x_f \in \mathbb{N} : \; x \geq x_f \; \Rightarrow f_{n+1}(x,x) > f(x).$$

Corolario 2.3.1 *De lo anterior se desprenden los resultados siguientes:*

1. *Para todo n: $\mathcal{E}^n \neq \mathcal{E}^{n+1}$.*
2. *La clase de las funciones elementales es un subconjunto propio de las recursivas primitivas.*
3. *El esquema de recursión acotada es insuficiente para definir a las funciones recursivas primitivas.*

2.4 Función de Ackermann

En la sección anterior presentamos una jerarquía para las funciones recursivas primitivas. Se vió que las funciones en un nivel de la jerarquía, que no se encuentran en el nivel anterior, tienen un crecimiento sustancialmente mayor que las funciones en el nivel anterior.

Extrapolaremos la construcción presentada para obtener una función con un crecimiento mayor que cualquier función en cualquier nivel de la jerarquía. Tal función no podrá ser recursiva primitiva, pero sí ha de ser computable. Veamos pues la construcción de esa función.

La función *sucesor* $g_0 = \text{suc} : x \mapsto x + 1$ es muy sencilla, desde el punto de vista computacional.

La función *suma* es una reiteración de la sucesor:

$$g_1 = \text{suc} : (x,y) \mapsto g_0^y(x).$$

La función *producto* es una reiteración de la suma:

$$g_2 = \text{pro} : (x,y) \mapsto g_1^{(y)}(x),$$

donde

$$\begin{aligned} g_1^{(1)}(x) &= x \\ g_1^{(y+1)}(x) &= g_1(x, g_1^{(y)}(x)) \end{aligned}$$

La función *exponencial* es una reiteración del producto:

$$g_3 = \exp : (x, y) \mapsto g_2^{(y)}(x).$$

Continuando con estas reiteraciones podemos considerar

$$g_{n+1} : (x, y) \mapsto g_n^{(y)}(x).$$

Esta es la motivación de la *función de Ackermann.*

Sea $A : \mathbb{N}^2 \to \mathbb{N}$ definida como sigue:

$$A(n, m) = \begin{cases} m + 1 & \text{si } n = 0, \\ A(n - 1, 1) & \text{si } n \neq 0 \text{ y } m = 0, \\ A(n - 1, A(n, m - 1)) & \text{si } n \neq 0 \text{ y } m \neq 0. \end{cases}$$

Para ilustrar el comportamiento de A, si escribimos $h_n(m) = A(n, m)$ entonces tendremos

$$h_{n+1}(m) = h_n^{m+1}(1), \ \ \forall m > 0$$

de lo cual resulta

$$\begin{aligned} h_0(m) &= m + 1 \\ h_1(m) &= m + 2 \\ h_2(m) &= 2m + 3 \\ h_3(m) &= 2^{m+3} - 3 \\ h_4(m) &= e_{m+3}(2) - 3 \end{aligned}$$

donde

$$e_1(x) = x \quad \text{y} \quad e_{y+1}(x) = x^{e_y(x)}.$$

Se ve así que esta función crece extremadamente rápido.

Una manera de calcular la función de Ackermann es manipulando listas de índices. Utilizaremos "corchetes" para denotar aplicaciones sucesivas, siempre por la derecha, de la función de Ackermann. Precisamente,

$$\begin{aligned} [n_1] &= n_1 \\ [n_1, n_2, \ldots, n_k] &= A(n_1, [n_2, \ldots, n_k]) \end{aligned}$$

Utilizaremos el símbolo "$\sim$" para denotar igualdad de listas bajo esta interpretación.

De la definición de la función tenemos

$$[n, m] \sim \begin{cases} [m + 1] & \text{si } n = 0, \\ [n - 1, 1] & \text{si } n \neq 0 \text{ y } m = 0, \\ [n - 1, n, m - 1] & \text{si } n \neq 0 \text{ y } m \neq 0. \end{cases}$$

Así pues, dada una lista de enteros $L \in \mathbb{N}^*$, separemos a sus dos últimos elementos, digamos

$$L = L' * [n, m]$$

entonces calculamos la transformación

$$T(L) = \begin{cases} L' * [m+1] & \text{si } n = 0, \\ L' * [n-1, 1] & \text{si } n \neq 0 \text{ y } m = 0, \\ L' * [n-1, n, m-1] & \text{si } n \neq 0 \text{ y } m \neq 0. \end{cases} \tag{2.9}$$

Tendremos el valor de $A(n, m)$ al obtener, por primera vez, una lista de longitud 1 aplicando sucesivamente la transformación T, a partir de la lista $[n, m]$.

En $\mathbb{N}^*$ introduzcamos la relación "$\leq_{\mathrm{A}}$" definiendo

$$\mathbf{m}_1 \leq_{\mathrm{A}} \mathbf{m}_2 \Leftrightarrow \exists n \geq 0 : T^n(\mathbf{m}_2) = \mathbf{m}_1.$$

Resulta evidente que

- $\leq_{\mathrm{A}}$ es una relación de orden,
- $\leq_{\mathrm{A}}$ es un buen orden, es decir toda cadena posee un mínimo, si y sólo si el procedimiento descrito para calcular la función de Ackermann termina.

Un momento de reflexión basta para convencerse de que $\leq_{\mathrm{A}}$ es, en efecto, un buen orden.

En la figura 2.1 se muestra un procedimiento alternativo para el cálculo de la función A. El seudocódigo ahí mostrado sigue una sintaxis *à la* FORTRAN.

Proposición 2.4.1 *La función de Ackermann NO es recursiva primitiva.*

Justificación: No entraremos en detalles. Tan solo diremos que debido a su definición, puede verse que la función $x \mapsto A(x, x)$ domina, a la larga, a cada una de las funciones $x \mapsto f_n(x, x)$, es decir,

$$\forall n \exists x_n : x \geq x_n \Rightarrow f_n(x, x) < A(x, x) \tag{2.10}$$

Si A fuese recursiva primitiva, entonces su diagonal $d_A : x \mapsto A(x, x)$ también sería recursiva primitiva, por lo cual existiría un índice n_0 tal que $d_A \in \mathcal{E}^{n_0}$. Por la proposición 2.3.7, se tiene $\forall x : d_A(x) < f_{n_0+1}(x, x)$. Esto contradice a la relación (2.10). □

```
Entrada: (m, n) ∈ ℕ².
Salida:  Z = A(m, n).

    inicio ;
    si m = 0 entonces { Z:= n + 1 ; fin}
    en otro caso
        Para cada  j ∈ [1, m + 1]: { valor[j]:= 1 ; lugar[j]:= − 1 }
1       valor[1]:= 1
        lugar[1]:= 0
2       valor[1]:= valor[1] + 1
        lugar[1]:= lugar[1] + 1
        i:= 1
(B):3   si lugar[i] = 1 ve hacia 5
        en otro caso
            si lugar[i] = valor[i + 1] ve hacia 4
            en otro caso ve hacia 2
4           valor[i + 1]:= valor[1]
            lugar[i + 1]:= lugar[i + 1] + 1
            si i = m ve hacia 6
            en otro caso
                i:= i + 1
                ve hacia 3
5       valor[i + 1]:= valor[1]
        lugar[i + 1]:= 0
        si i = m ve hacia 6
        en otro caso ve hacia 2
6       si lugar[m + 1] = n entonces { Z:= valor[1] ; fin}
        en otro caso ve hacia 2
```

Figura 2.1: Cálculo de la función de Ackermann.

Capítulo 3

Funciones de apareamiento y universalidad

Hemos visto ya que, utilizando la enumeración de Cantor, puede probarse que la colección de programas-**while** es numerable. En este capítulo realzaremos la propiedad de numerabilidad, haciendo abstracción de la enumeración particular que se tome en el conjunto de secuencias de números naturales. Presentaremos también, como un ejemplo particular de codificación de secuencias por números, a la función β de Gödel. Finalmente, adentrándonos en la propiedad de numerabilidad efectiva de los programas-**while**, veremos que existen programas-**while** universales, es decir, tales que dado el código de otro programa-**while** y el código de una secuencia de datos, entonces el programa universal aplica ese otro programa en el conjunto dado de datos.

3.1 Funciones de apareamiento

Estas funciones asocian, de manera única, un número natural a una pareja de naturales. Pueden ser consideradas funciones de codificación de parejas por números.

Diremos que una función $a : \mathbb{N}^2 \to \mathbb{N}$ es de *apareamiento* si ella es inyectiva. Para una tal función sus *proyecciones* son funciones $\pi_1^a, \pi_2^a : \mathbb{N} \to \mathbb{N}$ tales que

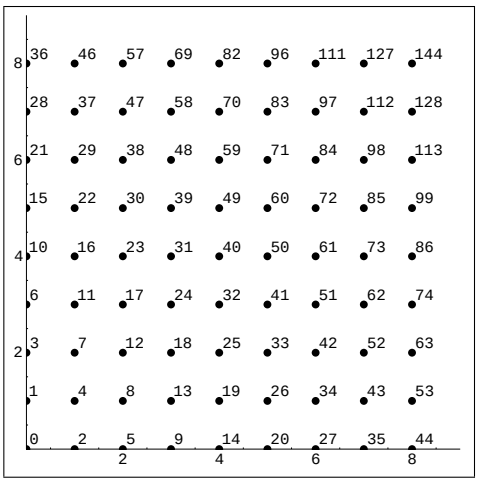
$$\begin{array}{rcl} \forall i,j \in \mathbb{N} & : & \pi_1^a(a(i,j)) = i \ \& \ \pi_2^a(a(i,j)) = j \\ \forall z \in \mathbb{N}^2 & : & a(\pi_1^a(z), \pi_2^a(z)) = z \end{array}$$

Hasta ahora la única función de apareamiento que hemos visto es la enumeración de Cantor, $c : (x,y) \mapsto \frac{1}{2}(x+y)(x+y+1)+x$. En la figura (3.1)-(a) mostramos la enumeración que hace de los puntos de $\mathbb{N}^2$. En las figuras (3.1)-(b) presentamos a la primera proyección $\pi_1^c : z \mapsto \pi_1^c(z)$, graficada primero para $z = 0, \ldots, 45$ y luego para $z = 0, \ldots, 55$ (observemos que 45 y 55 son números de la forma $\frac{1}{2}j(j+1)$, con $j = 9, 10$). En las figuras (3.1)-(c) presentamos a la segunda proyección $\pi_2^c : z \mapsto \pi_2^c(z)$, evaluada en los mismos dominios.

Proposición 3.1.1 *Si una función de apareamiento a es computable entonces sus proyecciones π_1^a, π_2^a también lo son.*

Dem. Observemos que la proposición se cumple considerando, en particular, la enumeración de Cantor $c : (x,y) \mapsto \frac{1}{2}(x+y)(x+y+1)+x$ que, de hecho, es computable y de apareamiento.

En efecto, sea $diag : \mathbb{N} \to \mathbb{N}$ la función que dado z indica "entre cuáles sumas de primeros números



(a)

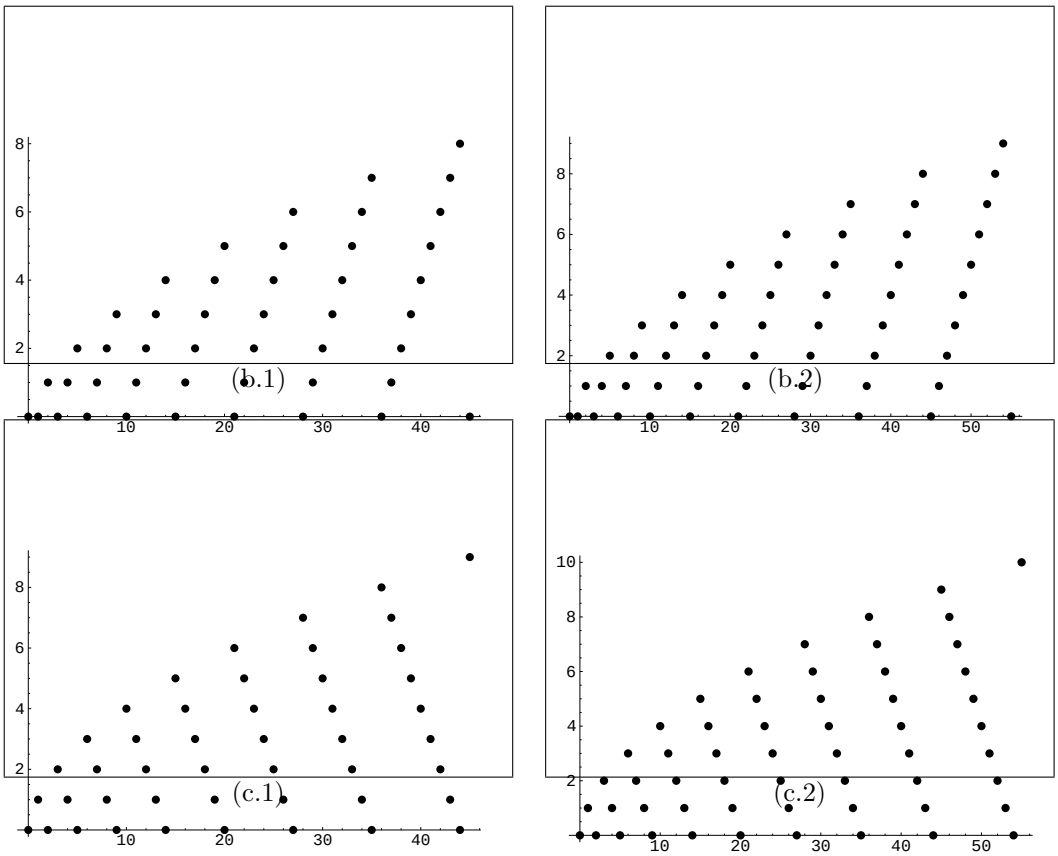


Figura 3.1: (a) Enumeración de Cantor. (b.1) Primera proyección hasta $z = 45$. (b.2) Primera proyección hasta $z = 55$. (c.1) Segunda proyección hasta $z = 45$. (c.2) Segunda proyección hasta $z = 55$.

naturales está z",

$$diag : z \mapsto \text{Min}\left\{ d | \frac{1}{2}(d)(d+1) \leq z < \frac{1}{2}(d+1)(d+2) \right\}.$$

Es evidente que *diag* es recursiva primitiva. Las proyecciones de la enumeración de Cantor son

$$\begin{aligned} \pi_1^c : z &\mapsto z - \frac{1}{2} diag(z)(diag(z)+1) \\ \pi_2^c : z &\mapsto diag(z) - \pi_1^c(z) \end{aligned}$$

Así pues, son también recursivas primitivas y, consecuentemente, computables.

Ahora, dada a, cualquier función de apareamiento y computable, podemos calcular sus proyecciones de la siguiente forma:

Dado $z \in \mathbb{N}$,

1. Pruébese, siguiendo la enumeración de Cantor, para cada pareja $\mathbf{x} \in \mathbb{N}^2$ si acaso corresponde a z bajo a.
2. La primera pareja hallada que cumpla con lo anterior determina a las proyecciones.

En símbolos:

Si $\eta : z \mapsto \text{Min}\{z_c \| a(\pi_1^c(z_c), \pi_2^c(z_c)) = z\}$, entonces
$\pi_1^a(z) = \pi_1^c(\eta(z))$ y $\pi_2^a(z) = \pi_2^c(\eta(z))$.

El algoritmo descrito siempre ha de terminar y, en consecuencia, a es computable y lo son también sus funciones proyecciones. □

3.1.1 Algunas funciones de apareamiento

Exceso de cuadrados

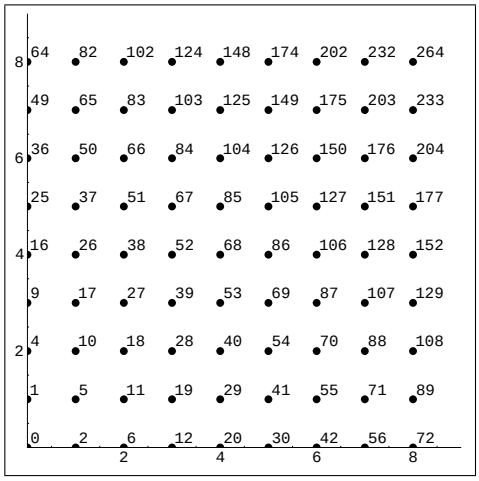
La función $ec : (x, y) \mapsto (x+y)^2 + x$, es una inyección, pues si tuviésemos que $ec(x, y) = z$, puesto que

$$(x+y+1)^2 = (x+y)^2 + 2(x+y) + 1 > (x+y)^2 + x = z$$

necesariamente hemos de tener que $(x+y)$ ha de ser la parte entera de la raíz cuadrada de z. Esto determina de manera única a x y también a y. De hecho, las proyecciones son

$$\begin{aligned} \pi_1^{ec} : z &\mapsto z - \left(\lfloor \sqrt{z} \rfloor\right)^2, \\ \pi_2^{ec} : z &\mapsto \lfloor \sqrt{z} \rfloor - \pi_1^{ec}(z) \end{aligned}$$

En la figura (3.2)-(a) mostramos la "enumeración" que hace *ec* de los puntos de $\mathbb{N}^2$. Aquí las comillas son pertinentes pues aunque *ec* es inyectiva, no es suprayectiva. Habrá algunos valores z para los cuales no existe pareja alguna $(x, y) \in \mathbb{N}^2$ tal que $z = ec(x, y)$ y esto es claro al seguir los valores consecutivos en la gráfica. En las figuras (3.2)-(b) presentamos a la primera proyección $\pi_1^{ec} : z \mapsto \pi_1^{ec}(z)$, graficada primero para $z = 0, \ldots, 36$ y luego para $z = 0, \ldots, 49$ (observemos que 36 y 49 son números de la forma j^2, con $j = 6, 7$). En las figuras (3.2)-(c) presentamos a la segunda proyección $\pi_2^{ec} : z \mapsto \pi_2^{ec}(z)$, evaluada en los mismos dominios. Aquí vemos que para algunos valores se tomaría valores negativos. En el contexto de nuestra actual presentación, el dominio de todas las funciones es el conjunto de los números enteros no-negativos. Así pues, en aquellos valores donde sea negativo el correspondiente valor "de la segunda proyección", ésta, de hecho, no está definida.



(a)

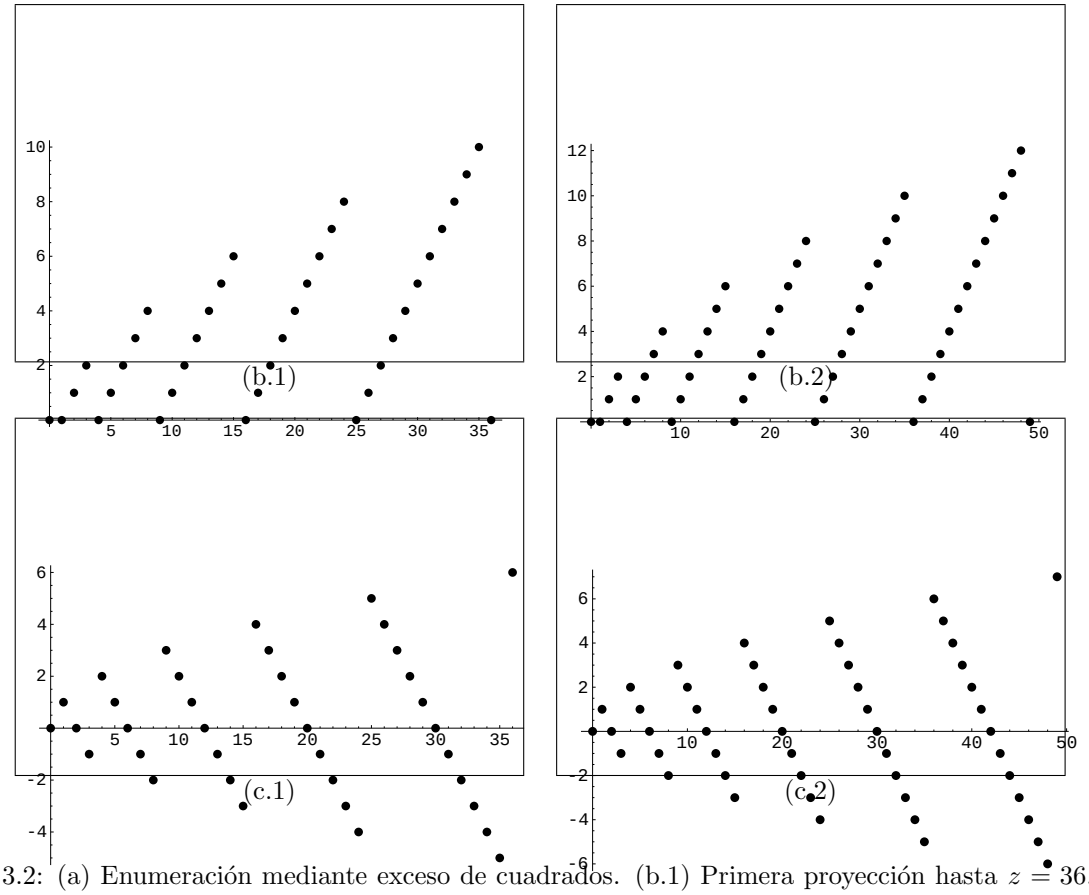


Figura 3.2: (a) Enumeración mediante exceso de cuadrados. (b.1) Primera proyección hasta $z = 36$. (b.2) Primera proyección hasta $z = 49$. (c.1) Segunda proyección hasta $z = 36$. (c.2) Segunda proyección hasta $z = 49$.

Exceso de potencias de 2

La función $ep : (x, y) \mapsto 2^x(2y + 1) - 1$, es también una inyección y sus proyecciones son

$$\begin{aligned} \pi_1^{ep} : z &\mapsto \text{Max}\{x : (2^x | z + 1) \wedge (2^{x+1} \not| z + 1)\}, \\ \pi_2^{ep} : z &\mapsto \frac{1}{2}\left(\frac{z+1}{2^{\pi_1^{ep}(z)}} - 1\right) \end{aligned}$$

es decir, la primera proyección da la máxima potencia de 2 que divide a $z + 1$, donde z es el número donde se evalúa.

En la figura (3.3)-(a) mostramos la enumeración que hace ep de los puntos de $\mathbb{N}^2$. Aquí, tenemos que ep es también suprayectiva. En las figuras (3.3)-(b) presentamos a la primera proyección $\pi_1^{ep} : z \mapsto \pi_1^{ep}(z)$, graficada primero para $z = 0, \ldots, 32$ y luego para $z = 0, \ldots, 64$ (observemos que 32 y 64 son números de la forma 2^j, con $j = 5, 6$). En las figuras (3.3)-(c) presentamos a la segunda proyección $\pi_2^{ep} : z \mapsto \pi_2^{ep}(z)$, evaluada en los mismos dominios.

Codificación con primos

Sean p, q dos números primos. Para la inyección $c_{pq} : (x, y) \mapsto p^x q^y - 1$, las proyecciones son

$$\begin{aligned} \pi_1^{c_{pq}} : z &\mapsto \text{Max}\{x : (p^x | z + 1) \wedge (p^{x+1} \not| z + 1)\}, \\ \pi_2^{c_{pq}} : z &\mapsto \text{Max}\{y : (q^y | z + 1) \wedge (q^{y+1} \not| z + 1)\}. \end{aligned}$$

En otras palabras, dado un número z, x es la máxima potencia de p que divide $z + 1$ y y es la máxima potencia de q que divide $z + 1$. En la figura (3.4) mostramos la "enumeración" que hace c_{pq} de los puntos de $\mathbb{N}^2$. Aquí, tenemos que c_{pq} no es suprayectiva. Para que un número esté en la imagen de esta función, en la descomposición en primos de ese número sólo han de aparecer los primos p, q. En la gráfica mencionada podemos observar que en el eje horizontal aparecen los números de la forma $p^x - 1$ y en el vertical aparecen los números de la forma $q^y - 1$.

3.2 Reiteración de funciones de apareamiento

Sea $a : \mathbb{N}^2 \to \mathbb{N}$ una función de apareamiento. Definimos inductivamente $a_n : \mathbb{N}^n \to \mathbb{N}$ como sigue

$$\begin{array}{lcccl} a_1 & : & x & \mapsto & x \\ a_{n+1} & : & (\mathbf{x}, x) & \mapsto & a(a_n(\mathbf{x}), x) \end{array}$$

Para cada $j \leq n$ la *j-ésima proyección* π_{nj} se define de manera que

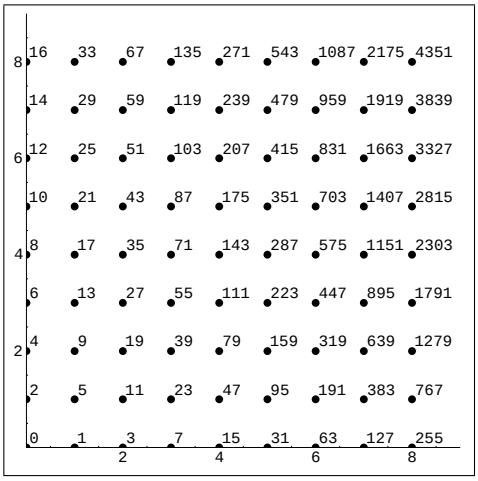
$$\forall \mathbf{x} \in \mathbb{N}^n : \quad \pi_{nj}(a_n(\mathbf{x})) = x_j.$$

Se ve fácilmente que $\forall z \in \mathbb{N}$:

$$\pi_{nj}(z) = \begin{cases} \pi_2^a \left(\pi_1^a\right)^{n-j}(z) & \text{si } j > 1, \\ \pi_1^a \left(\pi_1^a\right)^{n-2}(z) & \text{si } j = 1. \end{cases} \tag{3.1}$$

También podemos definir $a_* : \mathbb{N}^* \to \mathbb{N}$ como $\mathbf{x} \mapsto a\left(L(\mathbf{x}), a_{L(\mathbf{x})}(\mathbf{x})\right)$, donde $L(\mathbf{x})$ es la longitud de la secuencia $\mathbf{x}$. La inversa de a_* se calcula de manera inmediata: Dado $z \in \mathbb{N}$, $l = \pi_1^a(z)$ es la longitud del vector $\mathbf{x}$ el cual es, precisamente, $\mathbf{x} = (a_l)^{-1}(\pi_2^a(z))$. También, la función

$$\pi_{*j} : z \mapsto \begin{cases} x_j & \text{si } [j \leq n] \wedge [n = L(\mathbf{x})] \wedge [\mathbf{x} = (a_*)^{-1}(z)], \\ \perp & \text{en otro caso.} \end{cases} \tag{3.2}$$



(a)

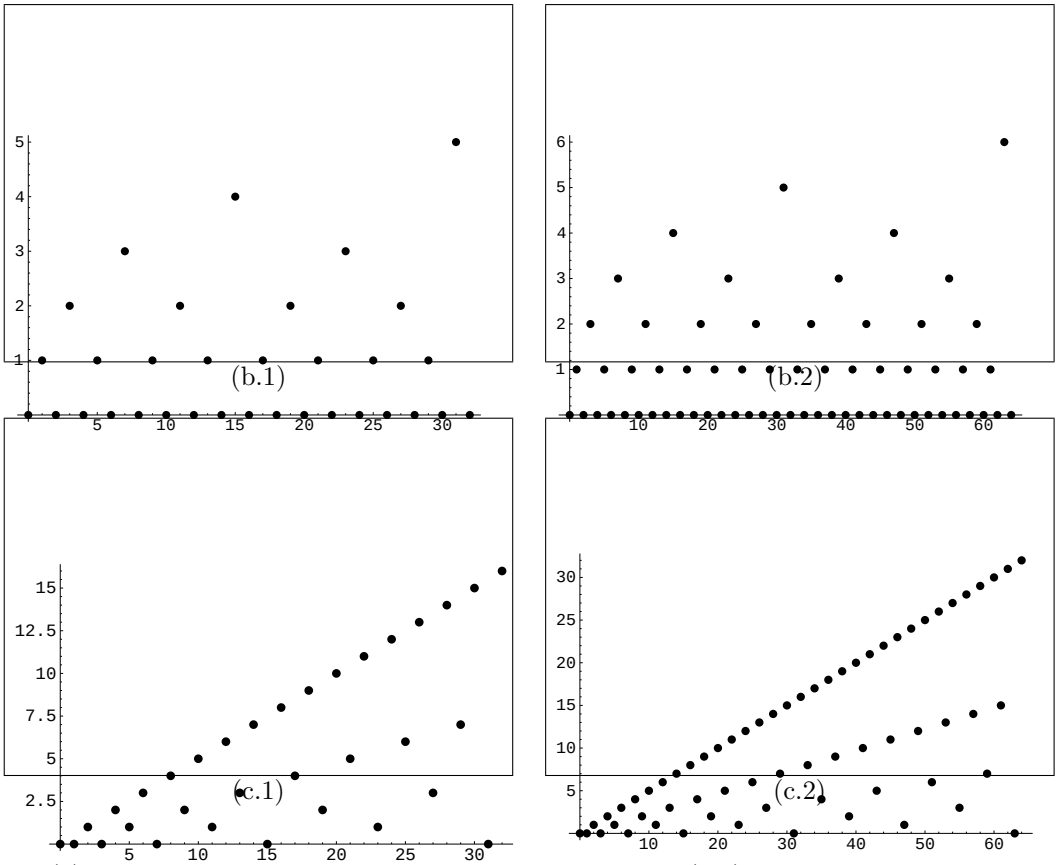


Figura 3.3: (a) Enumeración mediante exceso de potencias de 2. (b.1) Primera proyección hasta $z = 32$. (b.2) Primera proyección hasta $z = 64$. (c.1) Segunda proyección hasta $z = 32$. (c.2) Segunda proyección hasta $z = 64$.

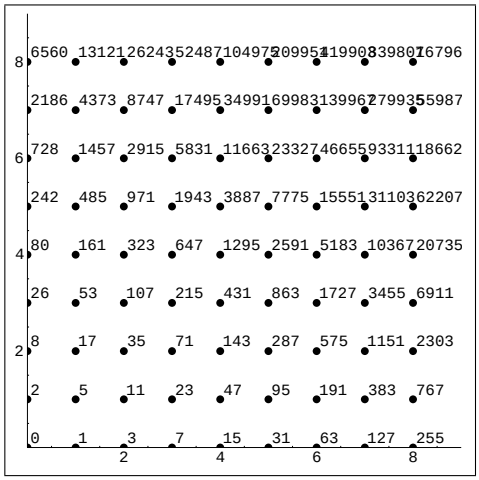


Figura 3.4: Enumeración mediante los primos $p = 2$ y $q = 3$.

en términos de las funciones π_{nj} puede escribirse como $\pi_{*j}(z) = \pi_{nj}(\pi_2^a(z))$, donde $n = \pi_2^a(z)$.

Para concluir esta sección, utilizando funciones de apareamiento computables, veremos que todas las funciones computables pueden ser calculadas poniendo cotas al número de variables que aparezcan en los programas-**while** que calculen a las funciones computables.

Observemos primeramente que se cumple la

Proposición 3.2.1 *Si la función de apareamiento a fuese computable por un programa-***while** *con a lo sumo m variables entonces su reiteración a_n ha de ser computable por un programa-***while** *con a lo sumo $m + n$ variables.*

Dem. Supongamos que P_a es un programa-**while** con m variables, de las cuales las dos primeras X, Y son de entrada y la última Z es de salida, que calcula $z = a(x, y)$ dada la pareja (x, y). Entonces la reiteración a_n se calcula por el programa siguiente con $n + m$ variables:

Input: $\mathbf{W} \in \mathbb{N}^n$: A n-vector given in a n-variables "array".
Output: $Z = a(\mathbf{W})$

$\{ \quad Z := W_1 \ ;$
$\quad Z := P_a(Z, W_2) \ ;$
$\qquad \vdots$
$\quad Z := P_a(Z, W_n)$
$\}$

Observamos que las n variables suplementarias se utilizan tan solo como almacenamiento de la "entrada" del programa descrito. □

Por otro lado, se tiene la

Proposición 3.2.2 *Se puede construir programas que calculen a las proyecciones con una cota uniforme para el número de variables utilizadas. Es decir:*

*Existe $p \in \mathbb{N}$ tal que para cualesquiera j, n que cumplan con la restricción $1 \leq j \leq n$, la j-ésima proyección π_{nj} se calcula por un programa-***while** *con a lo sumo p variables.*

DEM. De acuerdo con la ec. (3.1), la j-ésima proyección π_{nj} es una "reiteración" de a lo sumo $n-j+1$ proyecciones de a. Supongamos que Q_{1a} y Q_{2a} sean sendos programas-**while** que calculan a las proyecciones π_1^a y π_2^a. Para el cálculo de las proyecciones podemos proceder según el seudoprograma siguiente:

Input: $n, j \in \mathbb{N}$: Indexes of the required projection.
$z \in \mathbb{N}$: The instance of the projection function.
Output: $x = \pi_{nj}(z)$.

{ $z_1 = z$;
for $j_1 = 1$ **to** $n - j - 1$ **do** { $z_1 := Q_{1a}(z_1)$ } ;
if $j = 1$ **then** $x := Q_{1a}(z_1)$ **else** $x := Q_{2a}(z_1)$
}

p ha de ser el número de variables que resultan al escribir ese seudoprograma como un verdadero programa-**while**. □

De manera similar, puede verse que, si a es computable, entonces la función a_* también lo es.

Proposición 3.2.3 (Arreglos de variables) *Se puede dotar a las programas-***while** *de arreglos de variables.*

DEM. De la definición de a_* y de sus respectivas proyecciones en (3.2), tenemos que existe un programa-**while** Q_* tal que dados j, z calcula la j-ésima proyección $\pi_{*j}(z)$ como en (3.2).

Así pues, a un arreglo de variables $X[1:k]$ lo podemos representar por una variable $X1$ y asignaciones de la forma $X[j] := Z$ o $Z := X[j]$ quedarán como macros de los sendos seudoprogramas[1]:

{ $length := \pi_1^a(X1)$;
$Aux1 := \pi_2^a(X1)$;
$Aux2 := \pi_{length,1}(Aux1)$;
for $i = 2$ **to** $j - 1$ **do**
$\{Aux2 := a(Aux2, \pi_{length,i}(Aux1))\}$;
(*) $Aux2 := a(Aux2, Z)$;
for $i = j + 1$ **to** $length$ **do**
$\{Aux2 := a(Aux2, \pi_{length,i}(Aux1))\}$;
$X1 := Aux2$
}

{ $Aux1 := \pi_{*j}(X1)$;
(*) $Z := Aux1$
}

(en ambos seudoprogramas marcamos con un asterisco la instrucción "clave".) □

De aquí se tiene

Proposición 3.2.4 *Existe un entero $p \in \mathbb{N}$ tal que toda función computable $f : \mathbb{N}^n \to \mathbb{N}$ es calculada por un programa-***while** *con a lo sumo $(n + p)$ variables.*

DEM. Se puede utilizar funciones de apareamiento para "almacenar" los valores de las variables que excedan de $n + p$, donde p es como en la proposición anterior.

En efecto, a una variable se la puede considerar como un arreglo de todas las variables que hagan que el número $n + p$ se exceda. □

[1] Los programas presentados son meros ejemplos de que hay programas-**while** que realizan los procedimientos requeridos. De ninguna manera tratamos aquí noción alguna de *eficiencia* de programas.

3.3 Función β de Gödel

Esta función es una de las funciones de apareamiento más famosas en la historia de la lógica matemática. Kurt Gödel la utilizó para codificar fórmulas del cálculo de predicados por números. De esta forma pudo predicar sobre los mismos enunciados del cálculo de predicados. Tal propiedad de *autoreferencia* origina que, si se tuviese un sistema lógico consistente, codificable algorítmicamente en los número naturales, entonces el sistema ha de ser *incompleto*, vale decir, habrá enunciados que siendo universalmente válidos, no son demostrables. No es nuestro objeto estudiar aquí el teorema de incompletitud de Gödel y por tanto no abundaremos más sobre él. Sin embargo, tal tipo de argumentación lo hemos de utilizar en la presentación que hagamos de la noción de *irresolubilidad*.

La función β es muy sencilla desde el punto de vista de las funciones aritméticas y tiene un crecimiento moderado, comparado con otras funciones de apareamiento.

Comenzaremos presentando nociones básicas de la teoría de números, luego presentaremos el Teorema Chino del Residuo, después haremos la construcción de la función β de Gödel y, finalmente, mencionaremos la manera en la que codifica secuencias de números.

3.3.1 Nociones de divisibilidad

Recordamos la relación de *divisibilidad*. Dados $x, y \in \mathbb{Z}$ se dice que y *divide a* x, y se escribe $x|y$, si y sólo si $\exists z \in \mathbb{Z} : y = zx$.

Es claro que esta relación es

Reflexiva $\forall x : x|x$.

Transitiva $\forall x, y, z : (x|y) \wedge (y|z) \Rightarrow (x|z)$.

Antisimétrica (Salvo unidades) $\forall x, y : (x|y) \wedge (y|x) \Rightarrow \exists r \in \{-1, 1\} : y = rx$.

Consecuentemente, "|" es una relación de orden en el conjunto $\mathbb{Z}^+$, que consta de los enteros estrictamente positivos. Es claro que los elementos minimales de este orden son los números primos.

Para cada $n \in \mathbb{Z}$ el conjunto de *múltiplos* de n es $n\mathbb{Z} = \{y \in \mathbb{Z} : x|y\}$. Resulta que $n\mathbb{Z}$ es un *ideal* de $\mathbb{Z}$, es decir, se cumplen las propiedades siguientes:

$$\begin{array}{rrcl} & x, y \in n\mathbb{Z} & \Rightarrow & x + y \in n\mathbb{Z} \\ t \in \mathbb{Z}, & x \in n\mathbb{Z} & \Rightarrow & tx \in n\mathbb{Z} \end{array}$$

Dados $x, y \in \mathbb{Z}$ decimos que ellos son *congruentes módulo* n, y escribimos $x \sim_n y$, si $x - y \in n\mathbb{Z}$.

Resulta, ahora, que $\sim_n$ es una relación de equivalencia. El conjunto de clases de equivalencia, o *cociente*, es $\mathbb{Z}_n = (\mathbb{Z}/\sim_n) = \mathbb{Z}/n\mathbb{Z}$. En consecuencia se tiene que $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ es un anillo, pues de hecho, $\mathbb{Z}_n$ hereda la estructura de anillo de $\mathbb{Z}$.

Si $x \sim_n y$ escribimos, como es más usual, $x \equiv y \bmod n$.

Dados dos enteros $x, y \in \mathbb{Z}$ un *máximo común divisor* (m.c.d.) de la pareja x, y es un supremo del conjunto de cotas inferiores de $\{x, y\}$, respecto al orden de divisibilidad. Es decir, z es un m.c.d. de x, y si

$$\begin{array}{ll} 1) & (z|x) \wedge (z|y) \qquad \text{y} \\ 2) & \forall w : (w|x) \wedge (w|y) \Rightarrow w|z \end{array}$$

En tal caso escribimos $z = \text{m.c.d.}(x, y) = (x, y)$.

Dados $x, y \in \mathbb{Z}$ el *ideal generado* por x, y es el conjunto de combinaciones lineales de x e y; éste es $I(x, y) = \{ax - by | a, b \in \mathbb{Z}\}$.

Si w es un divisor común de x e y entonces w es también un divisor de cualquier elemento en $I(x,y)$.

Por tanto, el máximo común divisor (x,y) es el menor elemento positivo en el ideal generado por x,y. Así pues $\exists a_0, b_0 \in \mathbb{Z} : (x,y) = a_0 x - b_0 y$. Además esa combinación es mínima.

Algoritmo de Euclides

Este algoritmo permite calcular el máximo común divisor de dos números dados.

Dados $x, y \in \mathbb{Z}$ existen $q, r \in \mathbb{Z} : x = qy + r,\ 0 \leq r < y$. De hecho, la división común en los enteros da el *cociente* q y el *residuo* r, es decir, $q = x \text{ div } y,\ r = x \bmod y$.

El m.c.d, de x, y se calcula como sigue:

Sean $x_0 = x$, $x_1 = y$ y $\forall i \geq 2 : x_i = x_{i-1} \bmod x_{i-2}$.

Se tiene que $\{x_i\}_i$ es una sucesión de enteros no-negativos estrictamente decreciente. Por tanto, $\exists i_0 : x_{i_0} = 0$. El penúltimo elemento de la sucesión x_{i_0-1} es precisamente el m.c.d. de x, y.

En efecto, como $x_{i_0} = 0$ se tiene que $x_{i_0-1} | x_{i_0-2}$. Inductivamente se ve que $x_{i_0-1} | x_{i_0-j}, j = i_0-2, \ldots, 1, 0$.

Ahora, si w dividiese a x y a y entonces ha de dividir a x_0 y a x_1, luego divide también a x_2 y consecutivamente ha de dividir a x_{i_0-1}.

Para expresar a x_{i_0-1} como una combinación lineal de x_0, x_1 escribamos

$$x_{i_0-1} = a_j x_j - b_j x_{j+1},\ j = i_0 - 2, \ldots, 1, 0.$$

Entonces, necesariamente

$$\begin{array}{lcllcl}
a_{i_0-2} & = & 0 & b_{i_0-2} & = & -1 \\
 & \vdots & & & \vdots & \\
a_j & = & -b_{j+1} & b_j & = & -\left(a_{j+1} + b_{j+1}(x_j \text{ div } x_{j+1})\right) \\
a_{j-1} & = & -b_j & b_{j-1} & = & -\left(a_j + b_j(x_{j-1} \text{ div } x_j)\right) \\
 & \vdots & & & \vdots & \\
a_0 & = & -b_1 & b_0 & = & -\left(a_1 + b_1(x_0 \text{ div } x_1)\right)
\end{array}$$

Ejemplos. Consideremos dos ejemplos.

1. Para $x = 17711$ e $y = 10946$ el algoritmo de Euclides se desarrolla como se ve en la tabla 3.1.

De ahí, se obtiene que $1 = 4181 \cdot 17711 - 6765 \cdot 10946$.

En este ejemplo los números dados son dos términos consecutivos de la sucesión de Fibonacci. Es en estos casos que se obtiene los desarrollos "más largos" del algoritmo de Euclides.

2. Para $x = 7524482450817$ e $y = 499267608506$ el algoritmo de Euclides se desarrolla como se ve en la tabla 3.2.

De ahí, se obtiene que $1 = 185066460993 \cdot 7524482450817 - 2789144166880 \cdot 499267608506$.

Alternativamente, (x,y) se puede calcular factorizando a los argumentos como productos de números primos:

Si $x = \prod\{p^{exp(x,p)} | p \in \text{Primos}\}$ e $y = \prod\{p^{exp(y,p)} | p \in \text{Primos}\}$ entonces $(x,y) = \prod\{p^{z_p} | p \in \text{Primos}\}$ donde $\forall p : z_p = \text{Min}\{exp(x,p), exp(y,p)\}$.

Pasemos ahora a considerar algunas propiedades del máximo común divisor, visto éste como una operación binaria en $\mathbb{Z}^2$.

17711	=	1	·	10946	+	6765
10946	=	1	·	6765	+	4181
6765	=	1	·	4181	+	2584
4181	=	1	·	2584	+	1597
2584	=	1	·	1597	+	987
1597	=	1	·	987	+	610
987	=	1	·	610	+	377
610	=	1	·	377	+	233
377	=	1	·	233	+	144
233	=	1	·	144	+	89
144	=	1	·	89	+	55
89	=	1	·	55	+	34
55	=	1	·	34	+	21
34	=	1	·	21	+	13
21	=	1	·	13	+	8
13	=	1	·	8	+	5
8	=	1	·	5	+	3
5	=	1	·	3	+	2
3	=	1	·	2	+	1
2	=	2	·	1	+	0

Tabla 3.1: Algoritmo de Euclides para calcular (10946,17711)=1.

7524482450817	=	15	·	499267608506	+	35468323227
499267608506	=	14	·	35468323227	+	2711083328
35468323227	=	13	·	2711083328	+	224239963
2711083328	=	12	·	224239963	+	20203772
224239963	=	11	·	20203772	+	1998471
20203772	=	10	·	1998471	+	219062
1998471	=	9	·	219062	+	26913
219062	=	8	·	26913	+	3758
26913	=	7	·	3758	+	607
3758	=	6	·	607	+	116
607	=	5	·	116	+	27
116	=	4	·	27	+	8
27	=	3	·	8	+	3
8	=	2	·	3	+	2
3	=	1	·	2	+	1
2	=	2	·	1	+	0

Tabla 3.2: Algoritmo de Euclides para calcular (7524482450817,499267608506)=1.

$a\backslash x$	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
2	0	2	4	6	8	0	2	4	6	8
4	0	4	8	2	6	0	4	8	2	6
5	0	5	0	5	0	5	0	5	0	5
6	0	6	2	8	4	0	6	2	8	4
8	0	8	6	4	2	0	8	6	4	2
1	0	1	2	3	4	5	6	7	8	9
3	0	3	6	9	2	5	8	1	4	7
7	0	7	4	1	8	5	2	9	6	3
9	0	9	8	7	6	5	4	3	2	1

Tabla 3.3: Productos ax en $\mathbb{Z}_{10}$.

Proposición 3.3.1 (Operación mcd) *La operación* $(\cdot,\cdot) : \mathbb{Z}^2 \to \mathbb{Z}$, $(x,y) \mapsto \text{m.c.d.}(x,y)$ *es*

$$\begin{aligned} \textit{conmutativa} &: (x,y) = (y,x) \\ \textit{asociativa} &: ((x,y),z) = (x,(y,z)) \\ \textit{idempotente} &: (x,x) = x \\ \textit{posee un elemento anulador} &: (1,x) = 1 \end{aligned}$$

Esta operación no posee elementos neutros ni inversos.

Dos números se dicen ser *primos relativos* si $(x,y) = 1$. Es decir, si x e y no poseen factores comunes no-triviales.

Proposición 3.3.2 *La ecuación*

$$ax \equiv b \bmod n$$

tiene solución para todo b en $\mathbb{Z}_n$ si y sólo si a y n son primos relativos, es decir, $(a,n) = 1$.

Dem. $\Rightarrow$) Sea x_1 una solución para $b = 1$. Entonces

$$\begin{aligned} & ax_0 \equiv 1 \bmod n && \Rightarrow \\ \exists y : \quad & ax_0 - 1 = ny && \Rightarrow \\ & ax_0 - ny = 1 && \Rightarrow \\ & (a,n) = 1 \end{aligned}$$

$\Leftarrow$) Supongamos $(a,n) = 1$. Entonces $\exists x_0, y_0 : ax_0 - ny_0 = 1$.

Multiplicando por b: $a(x_0 b) - n(y_0 b) = b$. Sea $x_b = x_0 b$. Entonces $ax_b \equiv b \bmod n$.

Ejemplo. Ilustraremos aquí el efecto de la acción $x \mapsto ax$ para un n en particular y todos los posibles a.

Para $n = 10$, en la tabla 3.3 se muestran los valores de la homotecia $x \mapsto ax,\ a, x \in \mathbb{Z}_{10}$.

En su parte superior aparecen las "pendientes" a que no son primos relativos con 10. Vemos que para ellos, la homotecia no es suprayectiva y por tanto para algunos $b \in \mathbb{Z}_{10}$ no hay solución a la ecuación $ax \equiv b \bmod 10$. En la parte de abajo aparecen las "pendientes" a que sí son primos relativos con 10, se tiene como resultado, en cada renglón, una permutación de $\mathbb{Z}_{10}$ y por tanto la ecuación mencionada siempre tiene solución.

Proposición 3.3.3 *Si* $(m,n) = 1$ *entonces*

$$\forall x : \quad (m|x) \wedge (n|x) \ \Rightarrow\ mn|x.$$

El recíproco se cumple, naturalmente, para cualquier pareja de números.

DEM. Sea x tal que $(m|x) \wedge (n|x)$. Entonces

$$x = x_1 m \quad \& \quad x = x_2 n \ .$$

Luego,

$$\begin{array}{rcl} (m,n) = 1 & \Rightarrow & \exists a, b : am - bn = 1 \\ & \Rightarrow & x = axm - bxn = ax_2nm - bx_1mn = (ax_2 - bx_1)mn \\ & \Rightarrow & mn|x \end{array}$$

Corolario 3.3.1 *Si* $(m,n) = 1$ *entonces*

$$(x \equiv a \bmod m) \wedge (x \equiv a \bmod n) \ \Rightarrow \ (x \equiv a \bmod (mn)).$$

Proposición 3.3.4 *La siguientes dos condiciones son equivalentes*

- $(m,n) = 1$
- $\forall x : \ \ (m|xn) \ \Rightarrow \ m|x.$

DEM. Supongamos $(m,n) = 1$. Para algunos $a, b \in \mathbb{Z} : am - bn = 1$. Luego, para cualquier x: $axm - bxn = x$. Así pues, si $m|xn$ entonces también $m|x$ pues m divide a cada uno de los dos sumandos axm, bxn.

Recíprocamente, supongamos que $\forall x : \ \ (m|xn) \ \Rightarrow \ m|x$. Escribamos $d = (m,n)$. Entonces $\exists e_1, e_2 : m = e_1 d \ \wedge \ n = e_2 d$. Consecuentemente,

$$e_1 n = e_1(e_2 d) = e_2(e_1 d) = e_2 m,$$

o sea $m|e_1 n$ y por tanto $m|e_1$. Esto es posible sólo si $d = 1$.

3.3.2 Teorema Chino del Residuo

Teorema 3.3.1 *Si* $(m,n) = 1$ *entonces, cualesquiera que sean los números* $r, s \in \mathbb{Z}$ *el sistema de ecuaciones*

$$\begin{array}{rcl} x & \equiv & r \bmod m \\ x & \equiv & s \bmod n \end{array}$$

posee una solución x_0*. De hecho todo entero congruente con* x_0 *módulo* mn *es también una solución.*

DEM. Sean a, b tales que $am - bn = 1$. Multiplicando por $(r - s)$ tenemos

$$r - s = a(r - s)m - b(r - s)n.$$

Así que

$$r - a(r - s)m = s - b(r - s)n.$$

Denotemos por x_0 a ese valor común. Se tiene que x_0 es una solución del sistema de ecuaciones.

Iterando este resultado, obtenemos el siguiente

Teorema 3.3.2 *Sea* $[n_i]_{i=1,\ldots,k}$ *una sucesión finita de enteros positivos, primos relativos a pares:*

$$i_1 \neq i_2 \ \Rightarrow \ (n_{i_1}, n_{i_2}) = 1.$$

Sea $[r_i]_{i=1,\ldots,k}$ *una sucesión, de igual longitud, de enteros arbitrarios.*

$$
\begin{array}{rcl}
5 & \equiv & 1 \bmod 2 \\
5 & \equiv & 2 \bmod 3 \\
23 & \equiv & 5 \bmod (6 = 3 \cdot 2) \\
23 & \equiv & 3 \bmod 5 \\
53 & \equiv & 23 \bmod (30 = 5 \cdot 6) \\
53 & \equiv & 4 \bmod 7 \\
1523 & \equiv & 53 \bmod (210 = 7 \cdot 30) \\
1523 & \equiv & 5 \bmod 11 \\
29243 & \equiv & 1523 \bmod (2310 = 11 \cdot 210) \\
29243 & \equiv & 6 \bmod 13 \\
299513 & \equiv & 29243 \bmod (30030 = 13 \cdot 2310) \\
299513 & \equiv & 7 \bmod 17 \\
4383593 & \equiv & 299513 \bmod (510510 = 17 \cdot 30030) \\
4383593 & \equiv & 8 \bmod 19 \\
188677703 & \equiv & 4383593 \bmod (9699690 = 19 \cdot 510510) \\
188677703 & \equiv & 9 \bmod 23 \\
5765999453 & \equiv & 188677703 \bmod (223092870 = 23 \cdot 9699690) \\
5765999453 & \equiv & 10 \bmod 29
\end{array}
$$

Tabla 3.4: Ejemplo del Teorema Chino del Residuo.

Entonces existe una solución $x_0^k \in \mathbb{Z}$ *del sistema de ecuaciones*

$$x \equiv r_i \bmod n_i,\ i = 1, \ldots, k.$$

De hecho x_0^k *se calcula recursivamente: Si* x_0^{k-1} *es una solución de las primeras* $(k-1)$ *ecuaciones entonces* x_0^k *es una solución del sistema de ecuaciones*

$$
\begin{array}{rcl}
x & \equiv & x_0^{k-1} \bmod \left(\displaystyle\prod_{i=1}^{k-1} n_i \right) \\
x & \equiv & r_k \bmod n_k
\end{array}
$$

Ejemplos. Supongamos que **r** es la sucesión de los primeros diez enteros positivos y que **n** es la sucesión de los primeros diez números primos:

$$
\begin{array}{rcl}
\mathbf{r} & = & [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]^T \\
\mathbf{n} & = & [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]^T
\end{array}
$$

Procediendo recursivamente para calcular una solución del sistema de ecuaciones

$$x\mathbf{1} \equiv \mathbf{r} \bmod \mathbf{n} \qquad : \qquad x \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \equiv \begin{bmatrix} r_1 \\ \vdots \\ r_k \end{bmatrix} \bmod \begin{bmatrix} n_1 \\ \vdots \\ n_k \end{bmatrix}$$

obtenemos sucesivamente los valores mostrados en la tabla 3.4.

Así pues toda solución del sistema de ecuaciones es de la forma

$$x \equiv 5765999453 \bmod 6469693230.$$

3.3.3 Función β de Gödel

Definiciones básicas

La función β de Gödel se define como sigue:

$$\begin{array}{rcl} \beta : \mathbb{N}^3 & \rightarrow & \mathbb{N} \\ (x,y,i) & \mapsto & \beta(x,y,i) = x \bmod [1+(i+1)y] \end{array}$$

Se ve inmediatamente que valen las propiedades siguientes:

1. β es computable.
2. β es total.
3. $\forall (x,y,i) :\ 0 \leq \beta(x,y,i) < 1+(i+1)y$. Así pues, β tiene un crecimiento a lo sumo cuadrático.
4. Para cada x, y la función $i \mapsto \beta(x,y,i)$ es, a la larga, o sea, a partir de un punto, constante.

Codificación con la función β

Lema 3.3.1 *Los números $1+(i+1)n!$, $i = 0, \ldots, n-1$ son primos relativos a pares.*

Dem. Sean i, j tales que $0 \leq i < j \leq n$. Para ver que

$$(1+(i+1)n!, 1+(j+1)n!) = 1$$

basta probar, por la proposición 3.3.4, que $\forall x$:

$$1+(j+1)n! | x(1+(i+1)n!) \ \Rightarrow\ 1+(j+1)n! | x.$$

Para esto, observamos primero que, naturalmente, $(1+(j+1)n!, n!) = 1$, y $j-i | n!$.

Las siguientes implicaciones son entonces todas verdaderas:

$$\begin{array}{rl} & 1+(j+1)n! | x(1+(i+1)n!) = x(1+(j+1)n!) + x(i-j)n! \\ \Rightarrow & 1+(j+1)n! | x(i-j)n! \\ \Rightarrow & 1+(j+1)n! | x(i-j) \\ \Rightarrow & 1+(j+1)n! | xn! \\ \Rightarrow & 1+(j+1)n! | x \end{array}$$

quod erat demonstratum

Teorema 3.3.3 *Para cada sucesión finita de enteros $\mathbf{x} = (x_0, \ldots, x_{k-1}) \in N^*$ existen dos enteros x_0, y_0 tales que*

$$\forall i : 0 \leq i < k \ \Rightarrow\ \beta(x_0, y_0, i) = x_i.$$

Dem. Sea $n = \mathrm{Max}\left[\{x_i | i = 0, \ldots, k-1\} \cup \{k-1\}\right]$.

Para cualquier pareja de índices distintos i, j se tiene que los números $1+(i+1)n!, 1+(j+1)n!$ son primos relativos a pares. Por el Teorema Chino del Residuo resulta que

$$\exists z\ \forall i < n : z \equiv x_i \bmod (1+(i+1)n!).$$

suc'n	$(x_0$	,	$y_0 = n!)$
2	(6	,	2)
3	(287	,	6)
4	(361425	,	24)
5	(25359922214	,	120)
6	(139788730859742020	,	720)
7	(826489170740043812590949 07	,	5040)
8	(69854350964100746264124210002975731 55	,	40320)
9	(109111539031878053885632305958956831977973014841644	,	362880)
10	(39594118570427340632390268649843643728153543029317435961618201 6054	,	3628800)
11	40993304756811848300484593089803281189012676054379212562236873160 8\ (462609805556640065	,	39916800)
12	14589606272273774343473475388943631927183082214551025494836528024 0\ (90399203156187340223561847600306654727 7	,	479001600)
13	21166259352691195708355054485464559323411662068391573047395941692 3\ (91714282674183920236479170408001815819629748407204256000844890	,	6227020800)
14	14645929152270274834237394672156161277230356155692187864377850514 3\ 00282239890868199712422775496406456634391335784394208528831350917 8\ (2117190274044851456104	,	87178291200)
15	55910601798861848184323735180004397810763062353622200913785885871 9\ 89135275877580805893416176858481464523562338476166195116665116679 0\ (59468026105929962885855251334933119034602941440119	,1307674368000)

Tabla 3.5: Codificación mediante la función β de las sucesiones $\mathbf{n}$ con $n = 2, \ldots, 15$.

Tomamos pues $x_0 = z$ y $y_0 = n!$.

La pareja (x_0, y_0) es el *código* mediante la función β del vector $\mathbf{x}$. Escribiremos $(x_0, y_0) = cod_\beta(\mathbf{x})$.

Ejemplo. En la tabla 3.5 mostramos la codificación de los primeros quince segmentos de los números naturales,

$$\mathbf{n} = \{0, \ldots, n-1\}.$$

En la tabla 3.6 mostramos la decodificación de las parejas obtenidas en la tabla 3.5. El i-ésimo renglón corresponde a la i-ésima pareja (x_i, y_i) de la tabla 3.5. En la casilla correspondiente a la columna j-ésima aparece el valor $\beta(x_i, y_i)$, por tanto, en cada renglón n, aparece, hasta la posición $n-1$, la sucesión $\mathbf{n}$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	6	6	6	6	6	6	6	6	6	6	6	6	6	6
0	1	2	12	8	28	29	42	12	43	19	68	50	32	14	93
0	1	2	3	119	85	103	129	120	166	230	175	223	161	64	295
0	1	2	3	4	312	329	958	567	142	936	979	508	1523	1592	157
0	1	2	3	4	5	2101	1253	6461	331	368	2920	6238	6635	1386	1217
0	1	2	3	4	5	6	33228	34499	283	31760	3458	8542	24488	47683	45434
0	1	2	3	4	5	6	7	313460	336556	286535	412405	479975	419656	584730	152952
0	1	2	3	4	5	6	7	8	312771	3882988	202136	4467639	632324	3596573	656072
0	1	2	3	4	5	6	7	8	9	4459282	336886	16710305	22245956	25696698	30644117
0	1	2	3	4	5	6	7	8	9	10	439393614	37345292	402887647	119445100	118745010
0	1	2	3	4	5	6	7	8	9	10	11	4569489588	609356579	2409450031	3739331807
0	1	2	3	4	5	6	7	8	9	10	11	12	48749785550	88710030602	46048054024
0	1	2	3	4	5	6	7	8	9	10	11	12	13	462490916707	158842857626
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	19528279734488

Tabla 3.6: Decodificación de las parejas obtenidas en la tabla 3.5.

Codificación de secuencias

Consideremos la función $B_{*2} : \mathbb{N}^* \to \mathbb{N}^2$, $\mathbf{x} \mapsto B_{*2}(\mathbf{x}) = cod_\beta(\{L(\mathbf{x})\} * \mathbf{x})$ donde $\{L(\mathbf{x})\} * \mathbf{x}$ es el vector que se obtiene de anteponer al vector $\mathbf{x}$ su propia longitud, es decir,

$$\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{N}^n \;\Rightarrow\; \{L(\mathbf{x})\} * \mathbf{x} = (n, x_1, \ldots, x_n) \in \mathbb{N}^{n+1},$$

y consideremos también la función $B_{2*} : \mathbb{N}^2 \to \mathbb{N}^*$, $(x, y) \mapsto B_{2*}(x, y) = \mathbf{x} \in \mathbb{N}^n$ donde $n = \beta(x, y, 0)$ y $\forall i = 1, \ldots, n :\; x_i = \beta(x, y, i)$.

El lector se convencerá de inmediato de que se cumplen las relaciones siguientes:

1. B_{*2} es inyectiva, es decir: $B_{*2}(\mathbf{x}_1) = B_{*2}(\mathbf{x}_2) \Leftrightarrow \mathbf{x}_1 = \mathbf{x}_2$.
2. B_{*2} no es suprayectiva.

   En efecto, de acuerdo con la demostración del teorema 3.3.3, una pareja está en la imagen de cod_β sólo si su ordenada, es decir, su segunda componente, es el factorial de un número.
3. B_{2*} no es inyectiva.

   En efecto, de las tablas 3.5 y 3.6, vemos que las 14 parejas obtenidas en 3.5 son tales que bajo B_{2*} corresponden a la secuencia vacía, pues ésta es la única de longitud 0.
4. B_{2*} es suprayectiva, en consecuencia con el teorema 3.3.3.
5. $B_{2*} \circ B_{*2} = Id_{\mathbb{N}^*} : \mathbf{x} \mapsto \mathbf{x}$.
6. $B_{*2} \circ B_{2*} = Id_A : (x, y) \mapsto (x, y)$, donde $A \subset \mathbb{N}^2$ es la imagen, bajo la función cod_β, de $\mathbb{N}^*$.
7. Si $a : \mathbb{N}^2 \to \mathbb{N}$ es cualquier función de apareamiento (computable) entonces $a \circ B_{*2} : \mathbb{N}^* \to \mathbb{N}$ es una enumeración (computable), que no será suprayectiva, del conjunto de secuencias de números naturales.

3.4 Universalidad

3.4.1 Codificación de programas por números

Hemos visto que los programas-**while** se enumeran asociándole a cada símbolo $s \in A_{\mathbf{while}}$, donde $A_{\mathbf{while}}$ es el alfabeto de programas-**while**, un "dígito" $\lceil s \rceil \in [0, a_{\mathbf{while}} - 1]$, siendo $a_{\mathbf{while}} = \text{card}(A_{\mathbf{while}})$. A un programa $P = p_0 \cdots p_{m-1} \in A^*_{\mathbf{while}}$ se le asocia el "código" $\lceil P \rceil = \displaystyle\sum_{i=0}^{m-1} \lceil p_i \rceil a_{\mathbf{while}}^{m-(i+1)}$.

Con tal codificación de programas se cumplen las relaciones siguientes:

1. La imagen de $\lceil \cdot \rceil$ es un subconjunto propio y recursivo primitivo en $\mathbb{N}$.
2. $\lceil \cdot \rceil$ es una función inyectiva.
3. Dado el programa P se "calcula" procedimentalmente $n \in \mathbb{N}$ tal que $\lceil P \rceil = n$.
4. Dado n en la imagen de $\lceil \cdot \rceil$, se "calcula" procedimentalmente el programa P tal que $\lceil P \rceil = n$.
5. Sea $\{i_n\}_{n \geq 0}$ una enumeración recursiva primitiva de la imagen $\mathcal{P}$ de $\lceil \cdot \rceil$. Para cada programa-**while** P su índice es $n_P \in \mathcal{P}$ tal que $i_{n_P} = \lceil P \rceil$. Correspondientemente, escribiremos $P = P_{n_P}$. La función $P \mapsto n_P$ es una biyección efectiva $\{\text{programas-}\mathbf{while}\} \to \mathcal{P}$.

La enumeración $n \mapsto P$ es "la" enumeración efectiva de los programas-**while**. Ella cuenta a **todos** los programas-**while** y consecuentemente a todas las funciones computables.

Enunciamos esto como la

Proposición 3.4.1 (Numerabilidad efectiva) *La clase* {programas-**while**} *se puede poner en correspondencia biunívoca con un conjunto* $\mathcal{P} \subset \mathbb{N}$ *de manera tal que, dado* $P \in$ {programas-**while**} *se calcula procedimentalmente el índice* $n_P \in \mathcal{P}$ *correspondiente, y dado* $n \in \mathbb{N}$ *se puede decidir procedimentalmente si acaso existe* $P \in$ {programas-**while**} *tal que* $n = n_P$ *y, en tal caso, se puede también construir el programa* P.

Redefinimos aquí al operador $\lceil \cdot \rceil$, para escribir $n = \lceil P \rceil$, y recíprocamente, $\lfloor n \rfloor = P$, donde P y n guardan la relación expuesta en la proposición anterior.

Recordamos que una función $\mathbb{N}^n \to \mathbb{N}$ se dice ser *total* si ella está definida para cualquier $\mathbf{x} \in \mathbb{N}^n$.

En contraste con la proposición anterior, tenemos

Lema 3.4.1 (de la diagonal) *No existe una enumeración computable de todas las funciones computables totales.*

Demostración: Supongamos que hubiera una tal enumeración, digamos que la sucesión $\{f_n\}_{n\geq 0}$ enumera de manera efectiva, vale decir, computable, a las funciones computables totales.

Entonces para cualesquiera $n, m \geq 0$ el valor $f_{nm} = f_n(m)$ está bien definido, y, más aún, es computable a partir de la pareja (n, m).

La función $F : n \mapsto f_{nn} + 1$ es pues total y computable. Por tanto debe estar en la enumeración. Luego $\exists n_F : f_{n_F} = F$, es decir $\forall m : f_{n_F}(m) = F(m)$. En particular, para $m = n_F$ se tiene

$$f_{n_F}(n_F) = F(n_F) = f_{n_F n_F} + 1,$$

lo que implica que $0 = 1$. Esta contradicción muestra que no se puede enumerar efectivamente a todas las funciones totales.

3.4.2 Numerabilidad de la clase de máquinas de Turing

En el primer capítulo vimos que las funciones computables también pueden presentarse como funciones computables por máquinas de Turing.

En este sección veremos que las máquinas de Turing conforman una clase efectivamente numerable e introduciremos en términos de ellas el concepto de *universal*.

Sin pérdida de generalidad supondremos de aquí en adelante que el alfabeto de las máquinas de Turing es $A = \{0, 1\} =:$ Dos.

Al enumerar al conjunto de estados $E = \{q_i\}_{0\leq i\leq n-1}$, cada pareja de la forma $pt \in$ EA, donde p es un estado y t es un símbolo en S, se identifica con la cadena $it \in \text{Dos}^*$, donde i es el índice del estado p escrito en binario.

Recodifiquemos una cadena $x \in \text{Dos}^*$ de símbolos 0,1 aplicando la sustitución

$$0 \quad \to \quad 0 \quad ; \quad 1 \quad \to \quad 10 \tag{3.3}$$

y denotemos por $c(x)$ al resultado de la sustitución.

Así por ejemplo $c((9)_2 \cdot 1) = c(1001 \cdot 1) = 10001010$.

Codifiquemos a los símbolos de movimiento mediante la correspondencia

$$\text{Izqu} \quad \to \quad 110 \quad ; \quad \text{Dere} \quad \to \quad 1110 \quad ; \quad \text{Alto} \quad \to \quad 11110 \tag{3.4}$$

En 3.3 y en 3.4 hemos codificado 5 símbolos "originales[2]", a saber $0, 1, \text{Izqu}, \text{Dere}, \text{Alto}$, respectivamente por las cadenas 1^k0, $k = 0, 1, 2, 3, 4$. Conforme surja la necesidad de aumentar símbolos originales, los codificaremos por cadenas 1^k0 incrementando k según proceda.

[2]Utilizamos este adjetivo y evitamos caer en el simplismo de calificar con el sustantivo castellano "fuente" la forma inglesa "source".

Cada quíntupla de la forma $qsptm$ queda codificada por la yuxtaposición de $c(qspt)$, el código de m y de un separador de quíntuplas 1^50.

Una máquina de Turing $M = \{qs \to ptm\}_{qs \in \mathrm{EA}}$ queda entonces codificada por la concatenación de los códigos de las quíntuplas $qsptm$ que la describen, seguida de un separador de códigos de máquinas 1^60. Denotemos a este código por $\lceil M \rceil \in \mathrm{Dos}^*$, el cual puede ser visto como la representación en binario de un número natural. Llamemos a este número el *código de la máquina* M.

La codificación que hemos bosquejado es algorítmicamente reversible:

Dado un número natural $i \in \mathbb{N}$,

1. lo escribimos en binario,
2. si aparecen más de seis 1's consecutivos el número no representa una máquina de Turing, le podemos asociar la máquina *vacía*, NIL, i.e. la que no tiene quintupla alguna. En otro caso
3. decodificamos a las quíntuplas según el procedimiento inverso al arriba descrito. Si apareciese una inconsistencia, por ejemplo si apareciesen más quíntuplas que las necesarias, le asociamos la máquina vacía.

Tenemos así una enumeración $\{\text{máquinas de Turing}\} \to \mathbb{N}$ tal que para cualesquiera máquinas de Turing M_1 y M_2, que no sean NIL, se cumple la equivalencia lógica:

$$c(M_1) = c(M_2) \;\Leftrightarrow\; M_1 = M_2.$$

El conjunto imagen de las máquinas de Turing no-vacías bajo la codificación que hemos construído, y que denotamos por $N_{MT} = c(\{\text{máquinas de Turing}\} - \{\text{NIL}\}) \subset \mathbb{N}$, consta de los códigos "legales" de máquinas de Turing.

Como un subconjunto de los números naturales, N_{MT} es numerable y, consecuentemente, la clase de máquinas de Turing también lo es. Sea $\Psi_1 : \mathbb{N} \to N_{MT}$ definido como

$$\begin{aligned} \Psi_1(0) &= \mathrm{Min}\{m | m \in N_{MT}\} \\ \forall n \geq 0 : \Psi_1(n+1) &= \mathrm{Min}\,\{m | m \in N_{MT} - \{\Psi_1(0), \ldots, \Psi_1(n)\}\} \end{aligned}$$

Entonces $\Psi : M \mapsto \Psi_1^{-1} \circ c^{-1}(M) + 1$ es una biyección $(\{\text{máquinas de Turing}\} - \{\text{NIL}\}) \to (\mathbb{N} - \{0\})$, la cual, haciendo $\Psi : \text{NIL} \mapsto 0$, se extiende a una enumeración de todas las máquinas de Turing. Si $i = \Psi(M)$ diremos que i es el *índice de la máquina* M y escribiremos $M = M_i$, así como también, redefiniendo operadores, $i = \lceil M \rceil$ y $M = \lfloor i \rfloor$.

3.4.3 La noción de "Universalidad"

Para una clase de funciones dada, puede existir una función con un argumento más tal que ese argumento sea un parámetro para seleccionar todas y cada una de las funciones en esa clase. Tal función que parametriza a la clase se dice ser *universal*. De manera más formal:

Sea $\mathcal{F} = \{f_n\}_{n \geq 0}$ una clase numerable de funciones $\mathbb{N} \to \mathbb{N}$. La función $U_{\mathcal{F}} : (n, m) \mapsto f_n(m)$ se dice ser *universal* para $\mathcal{F}$. La clase $\mathcal{F}$ se *autorrepresenta* si $U_{\mathcal{F}} \in \mathcal{F}$.

Consideremos la enumeración efectiva de los programas-**while** $\Phi : n \mapsto P_n$ vista en la sección 3.4.1.

La función $U : (n, m) \mapsto \Phi(n)(m)$ es entonces universal para los programas-**while**.

Veremos que los sistemas que se autorrepresentan son formalmente *incompletos*, es decir:

- contendrán programas que no den ninguna respuesta en algunos puntos, es decir, que divergen en esos puntos, en otras palabras, hay programas que poseen un dominio incluído propiamente en su contraimagen, y
- dentro de ellos se puede plantear problemas que no son resolubles por ningún programa en tal sistema, es decir, se puede formular problemas irresolubles.

Para concluir la presente sección mostraremos el primer punto. Un programa, y consecuentemente la función que calcula, se dice ser *parcial* si diverge en algunos puntos.

Para esto, observamos que un programa universal en un sistema que se autorrepresenta hace la simulación de programas sobre datos manejando únicamente códigos. El uso de códigos es irrelevante, en el sentido de que es indiferente utilizar códigos o no, según se ve en el siguiente

Lema 3.4.2 (de Reescritura) *Si $h : \mathbb{N} \longrightarrow \mathbb{N}$ se calcula por un programa, entonces*

$$\exists\, n_h \in \mathbb{N}\ \forall n \in \mathbb{N} :\ \lfloor n_h \rfloor(\lfloor n \rfloor) = h(n). \tag{3.5}$$

En efecto, tenemos que las funciones de codificación pueden describirse en el sistema de programas y que éste es cerrado bajo la composición de funciones. Luego, dado el programa $h \in$ Programas-**while**, la función $g : m \mapsto h(\lceil m \rceil)$ es programable, por tanto posee un índice en el sistema, es decir $\exists\, n_h \in \mathbb{N} :\ \lfloor n_h \rfloor = g$.

Tenemos pues que $\forall n \in \mathbb{N}$ valen las igualdades siguientes: $\lfloor n_h \rfloor(\lfloor n \rfloor) = g(\lfloor n \rfloor) = h(\lceil \lfloor n \rfloor \rceil) = h(n)$.

Aquí me permito enfatizar que, mientras que del lado derecho de la ecuación en 3.5, tenemos el valor de h en el valor n, en su lado izquierdo, tenemos al valor de la n_h-ésima función computable aplicada sobre el n-ésimo posible argumento.

Lema 3.4.3 (de la Diagonal) *En todo sistema que se autorrepresenta existen funciones computables parciales.*

En efecto, al ser el sistema autorrepresentable, la función U es computable, luego la función $g : n \mapsto U(n, n) + 1$ es programable (obsérvese que $U(n, n)$ es el valor en n del n-ésimo programa).

Por el Lema de Reescritura se tiene que $\exists\, n_0 \in \mathbb{N}\ \forall n \in \mathbb{N} :\ \lfloor n_0 \rfloor(\lfloor n \rfloor) = g(n)$.

Es decir $\forall n \in \mathbb{N} :\ \lfloor n_0 \rfloor(\lfloor n \rfloor) = \lfloor n \rfloor(\lfloor n \rfloor) + 1$.

Así que para $n = n_0$, se ha de tener $\lfloor n_0 \rfloor(\lfloor n_0 \rfloor) = \lfloor n_0 \rfloor(\lfloor n_0 \rfloor) + 1$.

Esto obliga a que necesariamente $\lfloor n_0 \rfloor(\lfloor n_0 \rfloor) \uparrow$.

Así pues la n_0-ésima función es parcial.

3.4.4 Máquina Universal de Turing

Las máquinas de Turing constituyen el ejemplo por antonomasia de un sistema autorrepresentable. En la sección 3.4.2 vimos una codificación de máquinas de Turing. Esa la utilizaremos en esta sección.

Si la cinta de una máquina de Turing tiene inscrita la cadena $x \in \text{Dos}^*$, a ese contenido lo podemos codificar mediante la cadena $c(x) \in \text{Dos}^*$, donde c es la transformación descrita en la misma sección 3.4.2.

Así pues una pareja (M, x), donde M es una máquina de Turing y x es el contenido inicial de la cinta de M (que es propiamente el dato sobre el cual se aplica la máquina M), puede codificarse mediante la cadena $\lceil (M, x) \rceil = \lceil M \rceil \cdot 11111 \cdot c(x)$.

Una máquina de Turing MU se dice ser *universal* si su efecto es tal que

$$MU : \lceil (M, x) \rceil \mapsto \begin{cases} \lceil M(x) \rceil & \text{si } M(x) \downarrow, \\ \bot & \text{en otro caso} \end{cases}$$

en otras palabras, si MU "hace correr a la máquina M sobre el contenido x".

Proposición 3.4.2 *Existe una máquina universal de Turing.*

En efecto, es claro que la aplicación de quintuplas sobre la cadena x, o sus transformaciones sucesivas, es un procedimiento completamente descriptible dentro del paradigma de Turing. Por supuesto que es menester probar esta afirmación, y la manera de hacerlo es describiendo a la máquina universal. Sin embargo, dejamos al lector esta tarea como un ejercicio. Como mera guía mostramos a grandes pasos la construccción de una máquina universal de Turing, MUT.

MUT posee una cinta semi-infinita. En su primera parte ha de escribir el código de cualquier máquina M que ha de simular, llamemos a esta parte *área de programa*. Aparece luego el *área de trabajo*. Aunque esta porción de la cinta de MUT es semi-infinita, se puede suponer que se tiene la cinta infinita a ambos lados módulo la reducción de máquinas de Turing a máquinas con cinta semi-infinita vista en el primer capítulo. Así que cualesquiera movimientos en MUT han de interpretarse como movimientos módulo aquella reducción. Incorporemos en MUT a las quíntuplas necesarias para hacer corrimientos a la derecha y a la izquierda, y para sustituir una cadena por otra. Acaso en este punto será necesario introducir algunos "apuntadores" de corrimientos. Además introduciremos símbolos especiales para utilizarlos como delimitador izquierdo del contenido del área de trabajo, delimitador derecho de ese contenido, "cabeza lectora" de la máquina M y apuntador a la "quintupla activa" en el código de M.

Cuando comienza funcionar MUT, en su área de programa se encuentra el código de una máquina M y en el área de trabajo se encuentra el código de los datos $\mathbf{x}$ sobre los que MUT ha de aplicar el efecto de M.

El funcionamiento de MUT se resume más abajo. Reiteramos aquí que MUT trabaja sobre códigos. Aunque en la presentación nos referiremos a los símbolos originales, el lector ha de tener en cuenta que, en realidad, son los códigos de esos símbolos los objetos de los que se está hablando.

1. Al inicio se coloca la pareja $q_0 \cdot \langle CabLect \rangle (M)$ a la izquierda de $\mathbf{x}$. Tenemos pues una configuración del tipo $q_0 \cdot \langle CabLect \rangle (M) x_1 \mathbf{x}'$ en el área de trabajo, donde x_1 es el primer símbolo de $\mathbf{x}$ por la izquierda.

2. Mientras el área de trabajo sea de la forma $\mathbf{x}_{Izq} \cdot q \cdot \langle CabLect \rangle (M) s \cdot \mathbf{x}_{Der}$ y q no sea final, hágase lo siguiente,

   (a) Búsquese en el área de programa la quíntupla que comience con la pareja qs, digamos $qsptm$, la cual ha de ser la "quíntupla activa".

   (b) Sustitúyase la partícula $s_I \cdot q \cdot \langle CabLect \rangle (M) s \cdot s_D$, donde s_I es el primer símbolo de $\mathbf{x}_{Izq}$ por la derecha y s_D es el primer símbolo de $\mathbf{x}_{Der}$ por la izquierda, en el área de trabajo por la partícula $s_I \cdot p \cdot \langle CabLect \rangle (M) t \cdot s_D$.

   (c) Cámbiese esa última partícula por $p \cdot \langle CabLect \rangle (M) s_I \cdot t \cdot s_D$ o por $s_I \cdot t \cdot p \cdot \langle CabLect \rangle (M) s_D$, según $m =$ Izqu o $m =$ Dere, o bien declárese a q "final" si acaso $m =$ Alto.

Como un ejemplo, que no tiene aquí justificación alguna, hay una máquina universal de Turing, reportada en el excelente libro de Penrose, que tiene como índice al número que se muestra en las tablas 3.7 y 3.8, correspondientemente en decimal y en binario (Cfr. [27]).

Así pues la clase de máquinas de Turing es un sistema que se autorrepresenta, en el sentido definido en la sección 3.4.3.

Ahora, resulta evidente que la construcción bosquejada en la proposición 3.4.2, puede calcarse para construir una "máquina universal" con varias cintas que corra varios procesos en *tiempo compartido*. Es decir, consideremos $m+1$ cintas: En la primera, se recibe los códigos de m máquinas de Turing $M_1, \ldots, M_m$ seguidos del código de una lista de datos $\mathbf{x}$. Primero, en cada una de las cintas restantes, digamos la i, escribe $M_i \langle DelimMáq \rangle \mathbf{x}$. Luego, actúa como la m-ésima potencia cartesiana de la máquina universal, MUT^m, la

7244855335339317577198395039615711237952360672556559631
1081447966065050594042410903104836136323593656444345838222688332787676265561446928141177150178425517
0755408565768975334635694247848859704693472573998858228382779529468346052106116983594593879188554632
6440925525505820555989451890716536414896033096753020431553625034984529832320651583047664142130708819
3297172341510569802627346862299218381721537334828230734537134214750597403451843723595930906400243210
7734217885149276079759763441512307958639635449226915947965461471134570014504816733756217257346452273
1054482980784965126988788964569760906634204477989021914437932830019493570963921702904833270882596201
3017737272027186259199144282754374223513556751340842222998893744105343054710443686958764051781280194
3753081387063994277282315642528923751456544389905278079324114482614235728619311832661065612275553181
0207511085337633806031082361675045635852164214869542347187426437544428790062485827091240422076538754
2644541334517485662915742999095026230096337381377241621727477236102067868540028935660856968226201419
8248621698902609130940298570600174300670086896759034473417412787425581201549366393899690581773859165
4055356704092821332221631410978710814599786695997045096818419062994436560151454904880922084480034822
4920773040304318842989939313526688234966210194716191070146196852319284748203449589770955356110702658
1748733327296678998798473284098190764851272631001740166787363477605857245036964434897992034489997455
6624029374876688397514044516657077500605138839916688140725455446652220507242623923792115253181625125
36305093172863142200406457130527580230766518335199568913974813750492642960501001365198018694563949

Tabla 3.7: Código decimal de la máquina universal de Turing.

10000000010111010011010001001010101101000110100010100000110101001101000101010010110100001101000101
0010101101001001110100101001001011101010001110101010010010101110101010011010001010001010110100000110
1001000001010110100010011101001010000101011101001000111010010101000010111010010100110100001000011101
0100001110101000010010011101000101010110101001010110100000110101010010110100100100011010000000110100
0000111010100101010101110100001001110100101010101010101110100001010101110100001010001011101000101001
1010010000101001101001010010011010010001011010100010111010010010101110100101000111010100101001001110
1010101000011010010101010111010100100010110101000010110101000100110101010101000101101001010100100101
1101010010010111010101001010111010100101001101010100001110100010010010101110101010010101110101010000
0111010100100000110101010100101110101001010110100010010001110100000000111010010100101010101110100101
0010010101110100000010101110100001000111010000001010100111010000101001110100000010001011101000100001
1101000010010100111010001000010110100010100101110100010100101101001000001011010001010100100110100010
1010101110100100000011101001001010101011101010101001101001000101011010010010010110100000000101101000
0010001101000000100101101000000000011010010100010111010010101000110100101001010110100000100111010010
1010010110100100111010100000001010111010100000001101010100010101011010010101011010100001010111010100
1001010111010100010010110101001000010111010000000111010100100010110101001010011010101000101110101001
0100101110101010000001011101010100000101110100000001110101010000101011101001010101101010100001011101
0100010101011101010100100101110101010100001110101000000001110100100100001101001001000101101010101010
0111010000000010110100100001101010101010010111010010000110100100010101011101000010001110100010000111
0100001101000000101101000000100101110101010010101011010001000100101110100000100111010101001101000000
1010101101000010000111010010000100011101011010101001110100001001001110100010001000011101000010100101
1010000101000011101010101010101011101000100100110100010010011010100101001011101000100010101110100000
0001110100010010010111010011010010010000101101010101001101000101000101110100001101010000100010110101
0011010101001010010101101010100110100100101011101001101001000000101101000101010100011101001000010101
1010000000100110100100010010111010010000110101000000100101110100100101001101001001010101101001101001
0010100101101001101001010000001011010010000001110101001001101010101000010111010010100001011101001010
1010111010100010010110100100111010010101000101110100010011101010000101101001001110100101010101011101
0010001110100101010100101110100100011101010000001010101110011010100000101101001001110101000000010111
0100101101010000001010110100101001011101010000100101110100001101010001000010110101001101010001000101
1010101010100101110101000101001011010001010101010111010010000101011010100010111010100100101010111010
1010100100101110101000111010100000111010100100100101110101000111010100101000101110101000101110101000
0100101110101000111010001010001011101001010010111010100101010010111010010101010101011010100001010101
0110100001001110100001010101010111010100010101110101010001010111010000000111010101000100101110100000
0011101010100100010111010100000011010100001011010000000111010010000000101110101000111010100100010101
1101010011010101010100010101101000000101010100101010110100000001001101010101001001110101001101010101
0010010110101001101001001001110100000110101010101010010101101010001001101000101001010101110100000110
1010101010100101101000100011101000101010101010110100010001110100001010111010001001000011101001101000
0000010011101010000000100101110100010001010011101000000010010111010010101010010110100001010101011101
0001001010010111010000010001011101010100101101000100010011101000000100101011101000000010101011010000
1000111001111010000100000111010000100100111010000010100101110100000101001011010000100010101110100001
0001001101000100001110101111010000100100101110100001001001011101000000001010111010000101010001101000
1001011101000010000011101000010011101000100000101110101010010110100010000010111010000101010101110100
0000010101011101000100001010111010001000010101110100100000111010100100100110100000001010111010001000
1001011101010100001110101001010110100101010100001101000000101001101000000001110100000010010011101001
0110100100010100101101010100110100010100100101101010100110100010101010001011100110101001001011101010
1001101000101010101011001101010001010101100110100100010101010111010001000111010010010101010101110100
1010010100011010010000000101110100000110101010010101010110100101010101101001000100010111010001010101
1010100000101011010001000001101001000101011010000100111010100101010101011101001011010010010001010110
0110100100100101010111010011010010010010101101001011010010010010010110100101101001001010001011001101
0010010100101011101000101011101001001011100110100100101010010111001101001010001010101110100010001110
1000010100100101101001010001011101001010001010110100010011101001010001001011101000100111010010100010
0010111001101001000100011101000100111010010100101010111001101010010100000011100110101010101010110100
0000001110100101010010101011101001000111010010101001010111001101000010100100110011010101000001101000
0000011101001010101001010111001101010001000011010000000011010001001010101011101000100011101010101010
1010101101000010011101001000010010101110100101010001001101010000000010110100100111010100001010111010
0100001101010000000010110100100011101010010010111010000110101000010101011010100010111010100001010010
1011101010010111010100010101010111001101010001010101010000110101000010010101

Tabla 3.8: Código binario de la máquina universal de Turing.

cual, a grandes rasgos, tiene como conjunto de estados al producto $Q^m = \underbrace{Q \times \cdots \times Q}_{m \text{ veces}}$ y ante cualquier vector $(s_1, \ldots, s_m)$ leído de las cintas, si está en el estado $(q_1, \ldots, q_m)$ entonces actuará pasando al estado $(p_1, \ldots, p_m)$, y, para cada $i \leq m$ sustituye en la cinta i a s_i por t_i y hace el movimiento m_i, siempre que $(q_i s_i p_i t_i m_i)$ sea una quíntupla en MUT.

Ahora, la simulación anterior, puede realizarse con una máquina de Turing de una sola cinta, según se vió en el primer capítulo. Resulta entonces que se cumplen las siguientes dos proposiciones,

Proposición 3.4.3 (Tiempo compartido) *Para cada $m \geq 0$, existe una máquina de Turing MTC tal que para cualesquiera m máquinas de Turing $M_1, \ldots, M_m$ y para cualquier entrada $\mathbf{x}$ se tiene*

$$MTC(\mathbf{x}) = (M_1(\mathbf{x}), \ldots, M_m(\mathbf{x}))\,.$$

Proposición 3.4.4 (Operando "inter" de máquinas) *Para cada $m \geq 0$, existe una máquina de Turing MI tal que para cualesquiera m máquinas de Turing $M_1, \ldots, M_m$ y para cualquier entrada $\mathbf{x}$ se tiene que $MTC(\mathbf{x})$ asume el valor asumido por la máquina M_{i_0} en $\mathbf{x}$, si es que esa máquina es la primera, de entre todas las M_i's, en detenerse al actuar sobre la entrada $\mathbf{x}$, o bien queda indefinida, $MTC(\mathbf{x}) = \perp$, si todas las M_i's lo quedan con $\mathbf{x}$. La máquina MI se dice ser el* inter *de las M_i's y escribiremos $MI = \bigwedge_{i=1}^{m} M_i$.*

3.4.5 Universalidad en programas-while

Así como en el funcionamiento de las máquinas de Turing puede ser formalizado en las máquinas de Turing para dar así lugar a la máquina universal de Turing, tenemos que la sintaxis y la semántica de los programas-**while** pueden, también, ser formalizados en el lenguaje de los programas-**while**.

Al inicio de este capítulo presentamos una codificación de los programas-**while**. Esa codificación es programable, y dada una variable P siempre se puede decidir si su contenido es el código de un programa. El proceso de decisión conlleva un proceso de análisis sintáctico, que puede llevarse hasta el de una interpretación semántica del programa en cuestión.

Proposición 3.4.5 (Programa-while universal) *Existe un programa-***while** *PU tal que para cualquier $P \in$ Programas-***while** *y cualquier lista de datos $\mathbf{x}$ se tiene $PU(\lceil P \rceil, \lceil \mathbf{x} \rceil) = P(\mathbf{x})$.*

También, como las proposiciones 3.4.3 y 3.4.4 en el caso de las máquinas de Turing, resulta válida la

Proposición 3.4.6 *Existen sendos programas-***while** *P_1 y P_2 que calculan las funciones*

$$(\lceil p_1 \rceil, \ldots, \lceil p_m \rceil, \mathbf{x}) \mapsto (p_1(\mathbf{x}), \ldots, p_m(\mathbf{x})) \quad \text{y} \quad (\lceil p_1 \rceil, \ldots, \lceil p_m \rceil, \mathbf{x}) \mapsto \bigwedge_{i=1}^{m} p_i(\mathbf{x}).$$

Capítulo 4

Teoría de la recursividad

Presentamos en este capítulo a las funciones recursivas, las cuales han de coincidir con las funciones computables. Así, veremos una presentación alternativa de la noción de *computabilidad*.

Las funciones recursivas poseen una función universal y, en la clase que conforman, es irresoluble el "problema de la parada", es decir, el decidir cuándo el cálculo de una función recursiva ha de converger o no ante una instancia dada.

4.1 Funciones recursivas

Definición. La clase de las *funciones recursivas* es la mínima clase de funciones que contiene a las funciones recursivas primitivas y que es cerrada por el esquema de minimización (no necesariamente acotada).

Ejemplos. Los siguientes son ejemplos de funciones recursivas:

1. Todas las funciones recursivas primitivas son recursivas.
2. La función de Ackermann es recursiva pero no es recursiva primitiva.

   Efectivamente, en la sección 2.4 vimos que la función de Ackermann se calcula reiterando el operador T definido por la relación 2.9 de esa misma sección, hasta la primera vez que se obtenga una lista de longitud 1. El operador T es recursivo primitivo y la reiteración hasta la condición terminal se puede realizar mediante el esquema de minimización.

Teorema 4.1.1 *Las siguientes clases de funciones coinciden*

- *las funciones recursivas,*
- *las funciones computables,*
- *las funciones calculables por máquinas de Turing.*

En efecto, por un lado, toda función recursiva es computable pues las funciones básicas, a saber la operación 0, la sucesor y las proyecciones, son todas computables y los esquemas de composición, de recursión y de minimización son programables mediante programas-**while**.

Por otro lado, toda función computable es recursiva pues las funciones 0, sucesor y antecesor y la noción de asignación son representables mediante funciones recursivas. Luego, si P_1 y P_2 fuesen dos programas que calculen sendas funciones computables, entonces las funciones computadas por los programas $\boxed{\{P_1; P_2\}}$ y

$\boxed{\textbf{while } X \neq 0 \textbf{ do } P_1}$ son también representables mediante funciones recursivas, la última utilizando ciertamente el esquema de minimización.

Ya que la clase de funciones computables se autorrepresenta, tenemos como una consecuencia la siguiente

Proposición 4.1.1 *La clase de funciones recursivas se autorrepresenta, es decir, existe una función universal para esta clase que es en sí recursiva:*

$$\begin{array}{rcl} eval : N^2 & \rightarrow & N \\ (k,m) & \mapsto & f_k(m) : \text{el valor en } m \text{ de la } k\text{-ésima función recursiva.} \end{array}$$

El lema de la diagonal asevera que la subclase de funciones recursivas totales no se autorrepresenta.

4.2 Problema de la parada

Teorema 4.2.1 *No existe una función recursiva h tal que* $\forall (k,m)$:

$$h(k,m) = \begin{cases} 1 & \text{si } f_k(m)\downarrow, \\ 0 & \text{si } f_k(m)\uparrow. \end{cases}$$

Tal función h permitiría reconocer cuándo una función recursiva ha de converger para una instancia dada.

Demostración: Supongamos que la función h fuera recursiva. Podemos definir a partir de ella una función recursiva h_1 tal que

$$h_1(k,m) = \begin{cases} \perp & \text{si } f_k(m)\downarrow, \\ 0 & \text{si } f_k(m)\uparrow. \end{cases}$$

Entonces $\forall (k,m) : \ [h_1(k,m)\downarrow \Leftrightarrow \ f_k(m)\uparrow]$.

Su función diagonal $h_2 : k \mapsto h_1(k,k)$ es también recursiva. Para ella, $\forall k : \ [h_2(k)\downarrow \Leftrightarrow \ f_k(k)\uparrow]$.

Al ser h_2 recursiva posee un índice, digamos k_{h_2}. Entonces, $\forall k : \ \left[f_{k_{h_2}}(k)\downarrow \Leftrightarrow \ f_k(k)\uparrow\right]$.

En particular, al considerar $k = k_{h_2}$ hemos de tener $\left[f_{k_{h_2}}(k_{h_2})\downarrow \Leftrightarrow \ f_{k_{h_2}}(k_{h_2})\uparrow\right]$, lo que es claramente contradictorio.

4.3 Decidibilidad

En esta sección, dada una función f escribiremos, acaso de manera abusiva, "$f \in$ Programas-**while**" para indicar que la función f es *programable*, es decir, se calcula por un programa-**while**, o sea, f es computable, lo que equivale a decir que es recursiva.

4.3.1 Conjuntos decidibles

Dado un conjunto $A \subset N^m$ su función *característica* es

$$\chi_A : \mathbf{x} \mapsto \begin{cases} 1 & \text{si } \mathbf{x} \in A \\ 0 & \text{en otro caso.} \end{cases}$$

- A se dice ser *decidible* si $\chi_A \in$ Programas-**while**.
- A se dice ser *semidecidible* si $\exists\, g \in$ Programas-**while** : $A = \{\mathbf{x} | g(\mathbf{x})\downarrow\}$, es decir, si A es el dominio de alguna función programable.

A los conjuntos decidibles se les suele llamar también *recursivos* y a los semidecidibles, *recursivamente enumerables.*

Proposición 4.3.1 *A es decidible si y sólo si ambos A y su complemento $A^c = N^m - A$ son semidecidibles.*

En efecto, si A es decidible, entonces A y A^c son los respectivos dominios de las funciones

$$g_A : \mathbf{x} \mapsto \begin{cases} 1 & \text{si } \chi_A(\mathbf{x}) = 1, \\ \perp & \text{si } \chi_A(\mathbf{x}) = 0 \end{cases} \quad \text{y} \quad g_{A^c} : \mathbf{x} \mapsto \begin{cases} 1 & \text{si } \chi_A(\mathbf{x}) = 0, \\ \perp & \text{si } \chi_A(\mathbf{x}) = 1 \end{cases}$$

las cuales son efectivamente programables. Así que A y A^c son semidecidibles.

Recíprocamente, si A y A^c son semidecidibles y son los dominios de sendas funciones programables g_A y g_{A^c} entonces la característica de A coincide con la función

$$G : \mathbf{x} \mapsto \begin{cases} 1 & \text{si } g_A(\mathbf{x}) \downarrow, \\ 0 & \text{si } g_{A^c}(\mathbf{x}) \downarrow. \end{cases}$$

la cual, de manera similar a como se hizo en las proposiciones 3.4.4 y 3.4.6, resulta ser programable. Así que A es decidible.

Definamos a las clases de conjuntos decidibles y semidecidibles, en general y también en cada potencia cartesiana de N:

$$\begin{array}{lcl} \mathit{Recur} & = & \{A | A \text{ es decidible}\} \\ \mathit{Recur}_n & = & \mathit{Recur} \cap \mathcal{P}(N^n) \end{array} \quad \text{y} \quad \begin{array}{lcl} \mathit{RecEn} & = & \{A | A \text{ es semidecidible}\} \\ \mathit{RecEn}_n & = & \mathit{RecEn} \cap \mathcal{P}(N^n) \end{array}$$

donde para cualquier conjunto C, $\mathcal{P}(C)$ denota al conjunto de partes, o de subconjuntos, de C.

Observación 4.3.1 *Recur y RecEn son a lo sumo numerables.*

En efecto, cada conjunto decidible o semidecidible determina a una función computable y éstas, como ya hemos visto, forman un conjunto numerable.

Observación 4.3.2 $\forall\, n:$ *Recur_n forma un álgebra de conjuntos.*

Esto resulta de las igualdades

$$\chi_{A \cap B} = \chi_A \cdot \chi_B \ ; \ \chi_{A \cup B} = \chi_A + \chi_B - \chi_A \cdot \chi_B \ ; \ \chi_{A^c} = 1 - \chi_A$$

y de que las operaciones aritméticas son programables.

4.3.2 Proyecciones

Veremos en esta sección que la clase de conjuntos decidibles no necesariamente es cerrada bajo proyecciones. Esto dará en consecuencia que existen funciones no-computables que pueden ser formuladas, sin embargo, en el contexto de nociones de computabilidad.

Si $A \in \mathit{Recur}_n$ y $m \leq n$ la *proyección* de A en las primeras m coordenadas es

$$\Pi_m(A) = \{\mathbf{x} \in N^m | \exists \mathbf{y} \in N^{n-m} : (\mathbf{x}, \mathbf{y}) \in A\}.$$

Observación 4.3.3 *Toda proyección de un conjunto decidibles es un conjunto semidecidible, es decir*

$$A \in \mathit{Recur}_n \ \Rightarrow \ \Pi_m(A) \in \mathit{RecEn}_m.$$

En efecto, se tiene por definición $\left[\chi_{\Pi_m(A)}(\mathbf{x}) = 1 \Leftrightarrow \exists \mathbf{y} \in N^{n-m} : \chi_A(\mathbf{x}, \mathbf{y}) = 1\right]$, luego $\Pi_m(A)$ coincide con el dominio de la función programable $\mathbf{x} \mapsto \text{Min}\{\mathbf{y} | \chi_A(\mathbf{x}, \mathbf{y}) - 1 = 0\}$, donde el mínimo se toma según alguna enumeración (computable) de $\mathbb{N}^{n-m}$.

Esta última observación implica a su vez las siguientes dos relaciones ya demostradas:

$$\begin{aligned} A \in \mathit{Recur}_n &\Rightarrow A \in \mathit{RecEn}_n. \\ A \in \mathit{Recur}_n &\Leftrightarrow A, A^c \in \mathit{RecEn}_n. \end{aligned}$$

4.3.3 Problema de la parada (otra vez)

Para un conjunto $A \subset N^m$ consideremos el problema de *decisión* siguiente:

Problema Φ_A: Instancia: Un punto $\mathbf{x} \in N^m$.
Respuesta: $\begin{cases} \text{Sí} & \text{si } \mathbf{x} \in A, \\ \text{No} & \text{en otro caso.} \end{cases}$

Si A es un conjunto semidecidible el problema anterior es sólo resoluble a medias: Si g_A tiene como dominio a A entonces dado el punto $\mathbf{x}$ evaluamos en él a g_A. En el caso de que obtengamos un valor, contestamos que el punto pertenece a A pero en caso de no obtenerlo, no podremos saber si la falla en obtener valor alguno es debida a que efectivamente se carece de él o a que aún no concluye g_A su proceso de cálculo en $\mathbf{x}$. El problema de decisión en conjuntos semidecidibles es pues semidecidible.

Para un conjunto decidible, su propia función característica es una algoritmo de solución del problema de decisión correspondiente.

Así pues es altamente deseable contar con un procedimiento para identificar cuándo un programa ha de pararse para una entrada dada. Tal procedimiento es, sin embargo, inexistente.

Proposición 4.3.2 (Problema de la Parada) *Existen conjuntos semidecidibles que no son decidibles.*

En efecto, sea U un programa universal para la clase de programas. Consideremos el conjunto de parejas correspondientes a programas y entradas en sus dominios de convergencia:

$$\mathit{Defin} = \{(k, m) | U(k, m) \downarrow\}.$$

Veamos que $\mathit{Defin} \in \mathit{RecEn}_2$ pero $\mathit{Defin} \notin \mathit{Recur}_2$.

Consideremos el conjunto $\mathit{Buenos} = \{k | (k, k) \in \mathit{Defin}\}$ y su complemento $\mathit{Malos} = \{k | (k, k) \notin \mathit{Defin}\}$.

k es pues bueno si k mismo hace converger al k-ésimo programa. En otro caso es malo. Así pues

$$(k \in \mathit{Buenos} \Leftrightarrow \lfloor k \rfloor(\lfloor k \rfloor) \downarrow) \quad ; \quad (k \in \mathit{Malos} \Leftrightarrow \lfloor k \rfloor(\lfloor k \rfloor) \uparrow)$$

Si acaso se tuviera $\mathit{Defin} \in \mathit{Recur}_2$ entonces ambos conjuntos introducidos serían semidecidibles, es decir, $\mathit{Buenos}, \mathit{Malos} \in \mathit{RecEn}_2$.

Luego *Malos* sería el dominio de una función programable, es decir, $[\exists k_0 \forall k : \lfloor k_0 \rfloor(\lfloor k \rfloor) \downarrow \Leftrightarrow k \in \mathit{Malos}]$. En particular, para $k = k_0$ tendremos que ha de regir la equivalencia lógica $[k_0 \in \mathit{Buenos} \Leftrightarrow \lfloor k_0 \rfloor(\lfloor k_0 \rfloor) \downarrow \Leftrightarrow k_0 \in \mathit{Malos}]$ lo cual es, a todas luces, absurdo.

En conclusión: En cualquier sistema formal de cómputo que se autorrepresente podemos plantear el análogo al problema de la parada. Tal problema no podrá ser resuelto a menos de que el sistema en cuestión sea contradictorio, es decir, que haya un programa que ante una misma entrada terminará con una respuesta 0 o con una respuesta 1 indistintamente.

4.4 Teoremas de recursión

Pese a que los sistemas autorrepresentables adolecen de las incompletitudes antes vistas, ellos poseen propiedades de suma relevancia en la Teoría de la Computabilidad. Veremos que en ellos

- la "transmisión de parámetros" es un procedimiento programable,
- en cualquier reenumeración algorítmica de programas habrá un n-ésimo programa que es equivalente al n-ésimo programa de la enumeración canónica. En otras palabras, toda sucesión computable de programas poseee un "punto fijo", y
- existe un programa cuya única función es escribir su propio código. Dícese de tal programa que él "se autorreproduce".

Supondremos dada una función de codificación $\lceil\cdot\rceil : \{\text{programas}\} \to \mathbb{N}$ con inversa $\lfloor\cdot\rfloor : \mathbb{N} \to \{\text{programas}\}$.

Dos funciones f, g son *equivalentes*, $f \equiv g$, si ambas dan mismos resultados ante mismas instancias, es decir,

$$\forall \mathbf{x} : ([f(\mathbf{x}) \downarrow \Leftrightarrow g(\mathbf{x}) \downarrow] \wedge [f(\mathbf{x}) \downarrow \Rightarrow f(\mathbf{x}) = g(\mathbf{x})]) .$$

Teorema 4.4.1 (de Parametrización) *Existe una función programable h tal que*

$$\forall k \in N, \mathbf{x} \in N^n, \mathbf{y} \in N^m : \qquad \lfloor h(k, \mathbf{x}) \rfloor(\mathbf{y}) \equiv \lfloor k \rfloor(\mathbf{x}, \mathbf{y}).$$

En otras palabras, dado un programa g, cuyo índice es k, de dos listas de argumentos $\mathbf{x}$ y $\mathbf{y}$, entonces cuando se deja fijos a los valores de $\mathbf{x}$, es decir, cuando a éstos se les ve como *parámetros*, entonces la función resultante, con $\mathbf{y}$ como sola lista de argumentos, posee un índice que es función de k y de $\mathbf{x}$. Este teorema también se conoce como *Teorema s-m-n de Kleene* pues el matemático norteamericano Stephen Cole Kleene lo formuló llamando función s^n_m a la que aquí hemos llamado h.

La demostración del teorema consiste únicamente en ver que la expresión de la derecha es algorítmica:

Dados k y $\mathbf{x}$

1. decodificamos al programa $\lfloor k \rfloor$,
2. en él "instanciamos" los valores de las variables x's con los dados $\mathbf{x}$ y
3. codificamos el programa resultante.
4. El índice de ese programa se lo asignamos a $h(k, \mathbf{x})$.

Corolario 4.4.1 *Existe una función programable h_0 tal que $\lfloor h_0(x, y) \rfloor \equiv \lfloor \lfloor x \rfloor(y) \rfloor$.*

En efecto, consideremos la función $H : (x, y, z) \mapsto \lfloor \lfloor x \rfloor(y) \rfloor(z)$. Aquí, para hacer empatar a H con el teorema de paramatrización, hemos de considerar $n = m = 1$ ($k = x$, $\mathbf{x} = y$ y $\mathbf{y} = z$). El corolario se desprende inmediatamente del teorema de paramatrización.

Teorema 4.4.2 (de la Recursión) *Para cada $g \in$ Programas-**while** existe $k_g \in N$ tal que $\lfloor g(k_g) \rfloor \equiv \lfloor k_g \rfloor$.*

En efecto, sea h_0 tal que $\lfloor h_0(x, y) \rfloor = \lfloor \lfloor x \rfloor(y) \rfloor$.

Sea $h_1 : x \mapsto h_1(x) = g(h_0(x, x))$. Hagamos $k_1 = \lceil h_1 \rceil$ y $k_g = h_0(k_1, k_1)$. Se tiene la cadena de programas equivalentes siguiente:

$$\lfloor k_g \rfloor \equiv \lfloor h_0(k_1, k_1) \rfloor \equiv \lfloor \lfloor k_1 \rfloor(k_1) \rfloor \equiv \lfloor h_1(k_1) \rfloor \equiv \lfloor g(h_0(k_1, n_1)) \rfloor \equiv \lfloor g(k_0) \rfloor$$

4.5 Teoremas de autorreproducción

Teorema 4.5.1 (de Autorreproducción) *Existe un programa cuya función es escribir su propio índice, es decir,*

$$\exists\, k_0 \in N :\ \lfloor k_0 \rfloor : k \mapsto \lfloor k_0 \rfloor(k) = k_0.$$

En efecto, sea $\pi : (x, y) \mapsto x$. π es programable, i.e. $\pi \in$ Programas-**while**. Sea $k_0 = \lceil \pi \rceil$ su índice. Por el teorema de parametrización, $\exists\, h \in$ Programas-**while** $\forall k, x, y :\quad \lfloor h(k, x) \rfloor(y) = \lfloor k \rfloor(x, y)$.

La función $g : x \mapsto h(k_0, x)$ es programable. Por tanto, por el teorema de la recursión podemos encontrar un punto fijo para la enumeración "módulo" g. Sea $k_g \in N$ tal que $\lfloor k_g \rfloor \equiv \lfloor g(k_g) \rfloor$.

Se tiene entonces, $\forall\, m$:

$$\begin{array}{rcll} \lfloor k_g \rfloor(m) & = & \lfloor g(k_g) \rfloor(m) & : \text{por la elección de } k_g, \\ & = & \lfloor h(k_0, k_g) \rfloor(m) & : \text{por la definición de } g, \\ & = & \lfloor k_0 \rfloor(k_g, m) & : \text{por la elección de } h, \\ & = & k_g & : \text{porque } \lfloor k_0 \rfloor = \pi. \end{array}$$

Observación 4.5.1 *Un sistema autorrepresentable puede contener algunas otras funciones primitivas y contener también programas que se autorreproducen y realizan otras funciones.*

En efecto, sea $S = (Primi, RComp)$ un sistema de programación y sea $f : N \to N$ una función cualquiera. Sea $S' = (Primi', RComp')$ el sistema donde

$$\begin{array}{rcl} Primi' & = & Primi \cup \{f\} \\ RComp' & = & \{(f, \rho) | \rho \in RComp\}. \end{array}$$

Así todo programa en S' consta de un programa en S con la función f adjunta: $g' \in S' \ \Leftrightarrow\ \exists g \in S :$ $g' = (f, g)$.

Pues bien, puede verse que vale la siguiente

Proposición 4.5.1 *S' es un sistema autorrepresentable siempre que S lo sea.*

Consecuentemente S' posee una función "autorreproductora", sólo que en S' el programa que se autorreproduce

- realiza la función primitiva f, y
- crea una "copia" de sí mismo

Si k_0 es el índice de un programa autorreproductor en S', el programa $\lfloor k_0 \rfloor$ es propiamente, lo que en la actualidad llamamos, un "virus".

4.6 Virus en programas-while

La existencia de un programa que se autorreproduce fue demostrada por von Neumann[1] en la década de los años 40. Aunque desde ese entonces se vislumbraba la posibilidad de tener programas errantes, es sólo a mediados de los 70 que aparecen los virus en computadoras. En esta sección haremos una presentación más de tipo "programacional" de los virus, en el marco de la Teoría de la Computabilidad. Esta presentación no es, ni mucho menos, una introducción a los métodos de creación de virus computacionales.

[1] John von Neumann (1903-1957), matemático norteamericano de origen húngaro. Desde 1930 se incorporó a la Universidad de Princeton. Fue uno de los principales investigadores en el proyecto de la bomba atómica en Los Alamos. En 1952 diseñó la MANIAC I, que fue la primera computadora en el mundo con un programa flexible.

4.6.1 Definiciones básicas

Sea $\{P_i\}_{i\geq 0}$ una enumeración efectiva de los programas-**while**. Para cualesquiera $i, j \geq 0$ definimos

$$\begin{aligned} f_i = f_{P_i} &: \text{la función computada por el } i\text{-ésimo programa-}\mathbf{while}, \\ f_i(j) &: \text{valor de la } i\text{-ésima función en la } j\text{-ésima entrada}, \end{aligned}$$

Consideremos las siguientes tres funciones

$$\begin{aligned} \mathit{Rep} &: \mathbb{N} \to \{\text{programas-}\mathbf{while}\};\ \ i \mapsto P_i \\ &: \text{dado } i \text{ construye el } i\text{-ésimo programa-}\mathbf{while}, \\ \mathit{Comp} &: \mathbb{N}\times\mathbb{N} \to \mathbb{N};\ \ (i,j) \mapsto k = \mathit{Comp}(i,j) \\ &: \text{dados } i,j \text{ el valor } k \text{ dado por esta función es el índice del programa obtenido al "componer" } P_i \text{ con } P_j. \text{ Es decir, } P_k \text{ tiene como variables de entrada a las de entrada de } P_i \text{ y como variables de salida a las de salida de } P_j, \text{ ante una una instancia dada de entrada, } \mathbf{x}, P_k \text{ "corre" } P_i, \text{ aplica las variables de salida de } P_i \text{ en las de entrada de } P_j \text{ y entonces "corre" a } P_j. \\ \mathit{Eva} &: \mathbb{N}\times\mathbb{N} \to \mathbb{N};\ \ (i,j) \mapsto f_i(j) \\ &: \text{dados } i,j \text{ "simula" el cálculo de } f_i \text{ sobre } j. \end{aligned}$$

Resulta evidente que estas tres funciones son programables.

Un *virus* es un programa $P = P_m$ tal que $P(y) = \mathit{Rep}(m)$ para todo $y \in \mathbb{N}$. Así pues, un virus P es un programa que se autorreproduce.

El Teorema de Autorreproducción puede, pues, parafrasearse como la existencia de virus.

4.6.2 Matrices infinitas

Para mostrar que existen virus en el lenguaje de los programas-**while** necesitaremos arreglos planares infinitos, es decir, matrices cuyos renglones y columnas están siendo indicadas por los números naturales.

Una *matriz infinita* es un arreglo $M = [m_{ij}]_{i,j\geq 0}$. Denotemos a las matrices infinitas con entradas naturales y con entradas funciones recursivas, respectivamente, como

$$\begin{aligned} \mathbb{N}^{\mathbb{N}\times\mathbb{N}} &= \{M | M \text{ es matriz infinita con entradas en } \mathbb{N}\}, \\ \text{recursivas}^{\mathbb{N}\times\mathbb{N}} &= \{M | M \text{ tiene funciones recursivas como entradas}\}. \end{aligned}$$

Sea $f : \mathbb{N} \to \mathbb{N}$ una función y M una matriz. M se dice ser

cerrada por renglones bajo la función f si $\forall i_1 \exists i_2 \forall j :\ (f(m_{i_1 j}) = m_{i_2 j})$, es decir, si para cada renglón en M, su imagen bajo la función f aparece también como otro renglón de la matriz M, y

diagonalmente completa si su diagonal aparece también como un renglón, es decir $\exists i_d \forall j :\ (m_{i_d j} = m_{jj})$.

Lema 4.6.1 (Diagonal de punto fijo) *Si f es una función tal que existe una matriz diagonalmente completa cerrada por renglones bajo f entonces f posee un punto fijo, es decir, existe x tal que $f(x) = x$.*

Demostración: Sea M una matriz diagonalmente completa cerrada por renglones bajo f. Sea i_d tal que $\forall j : m_{i_d j} = m_{jj}$. Existe i_e tal que $\forall j : f(m_{i_d j}) = m_{i_e j}$. Para $j = i_e$ se tiene $f(m_{i_d i_e}) = m_{i_e i_e} = m_{i_d i_e}$. Así pues $m_{i_d i_e}$ es un punto fijo de f.

4.6.3 Teoremas de recursión

Una función $g : \mathbb{N} \to \mathbb{N}$ se dice ser *extensional* si $\forall i, j : \quad [f_i = f_j \;\Rightarrow\; f_{g(i)} = f_{g(j)}]$.

Ejemplos

1. Cualquier función constante es extensional.

2. Sea g la función construída como sigue: Dado i, calculamos el i-ésimo programa

$$P_i = P_i(X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, \ldots).$$

$g(i)$ será el código del programa obtenido al "instanciar" las primeras tres variables por los valores consecutivos 1,2,3 y renombrar las demás variables:

$$g(i) = \lceil P_i(X_0 \leftarrow 1, X_1 \leftarrow 2, X_2 \leftarrow 3, X_0 \leftarrow X_3, X_1 \leftarrow X_4, \ldots)\rceil.$$

Es evidente que g es extensional.

3. La función sucesor no es extensional: Si P_i y P_j son dos programas que calculan una misma función, no necesariamente P_{i+1} y P_{j+1} han de calcular una misma función.

Observación 4.6.1 *Toda función extensional g define un operador entre funciones recursivas:*

$$\begin{array}{rcl} \Gamma_g : \{\text{recursivas}\} & \to & \{\text{recursivas}\} \\ f_i & \mapsto & \Gamma_g(f_i) = f_{g(i)} \end{array}$$

Teorema 4.6.1 (Débil de Recursión) *Si g es extensional entonces Γ_g posee un punto fijo.*

Demostración: Sea M la siguiente matriz consistente de funciones recursivas, $\forall i, j : \quad m_{ij} = f_{f_i(j)}$. Si $f_i(j) = \perp$ entonces la función $f_{f_i(j)}$ coincidirá con la función *divergente en todo punto*: $\perp : x \mapsto \perp$.

M es efectivamente computable. De hecho, dados i, j procedamos como sigue:

1. con *R*ep calculamos P_i,

2. con *eval* calculamos $k = P_i(j)$,

3. tras calcular k, con *R*ep calculamos P_k.

Sea P^M el programa-**while** que implementa el procedimiento anterior. Entonces $P^M(i, j) = m_{ij}$.

M es diagonalmente completa: Sea Q tal que $Q(j) = P^M(j, j)$. Q es evidentemente "realizable" como un programa-**while** y consecuentemente $\exists i_d : P_{i_d} = Q$. Así pues

$$m_{i_d j} = P_{i_d}(j) = Q(j) = P^M(j, j) = m_{jj}.$$

M es cerrado por renglones bajo Γ_g: Dado i sea $P^{M,i;g}$ el programa tal que $\forall j : P^{M,i;g}(j) = P^M(g(i), j)$. Existe entonces $h(i) \in \mathbb{N}$ tal que $P_{h(i)} = P^{M,i;g}$. Entonces para cualquier j se tiene

$$m_{h(i)j} = P_{h(i)}(j) = P^{M,i;g}(j) = P^M(g(i), j) = m_{g(i),j} = \Gamma_g(m_{ij}).$$

Del Lema Diagonal de Punto Fijo se sigue que Γ_g posee un punto fijo.

Teorema 4.6.2 (De Recursión) *Si g es cualquier función recursiva entonces Γ_g posee un punto fijo, es decir $\exists k : \; f_k = f_{g(k)}$.*

Demostración: Dados i_1, i_2 diremos que f_{i_1} *se relaciona con* f_{i_2} *según* g, $f_{i_1} \overset{g}{\rightsquigarrow} f_{i_2}$, si existe k tal que

$$f_k = f_{i_1} \quad \& \quad f_{g(k)} = f_{i_2}.$$

Se tiene que $\forall i_1 \exists i_2 : f_{i_1} \overset{g}{\rightsquigarrow} f_{i_2}$.

Como M es además diagonalmente completa tenemos que algún elemento en su diagonal m_{ii} se relaciona consigo mismo. Así pues existe k tal que $f_k = m_{ii} = f_{g(k)}$.

4.6.4 Construcción de virus

Sea r el índice del programa *Rep*.

Dado $x \in \mathbb{N}$ sea P_x un programa-**while** que hace lo siguiente:

1. calcula $x_r = \mathit{Comp}(r, x)$, es decir, compone a *Rep* con el x-ésimo programa,
2. deja a x_r como su valor final.

P_x es un programa que calcula un valor constante, x_r, que depende de x.

Sea $g : \mathbb{N} \to \mathbb{N},\ x \mapsto$ (índice de P_x). Evidentemente, $\forall x, y$: $f_{g(x)}(y) = \mathit{Comp}(r, x)$. La función g no tiene porqué ser extensional, sin embargo es computable. Por el Teorema de Recursión se tiene que $\exists n_0$ tal que $f_{n_0} = f_{g(n_0)}$. Así pues, ambos f_{n_0} y $f_{g(n_0)}$ calculan a la función $y \mapsto \mathit{Comp}(r, n_0)$.

Sea $P = \mathit{Rep}(\mathit{Comp}(r, n_0))$. Se ve fácilmente que P es un virus.

Por un lado, tenemos que el índice de P es $\mathit{Comp}(r, n_0)$. Por otro lado, para todo y se ha de tener $P(y) = \mathit{Comp}(r, n_0)$.

Observación 4.6.2 *Las siguientes propiedades se siguen inmediatamente:*

1. *Todo lenguaje de programación suficientemente capaz de contener un "intérprete" de él mismo (de programar las funciones* Rep*,* Comp *y* Eval*) es susceptible de "engendrar" virus.*
2. *El virus se "realiza" como un punto fijo de una función de sustitución g. Esta es la base de los programas virus "reales".*
3. *El procedimiento descrito en esta sección,*
   - *es efectivo: en principio se puede escribir un programa que calcule a los índices r y k_0, así como también a los valores $\mathit{Comp}(r, k_0)$ y $\mathit{Rep}(\mathit{Comp}(r, k_0))$,*
   - *NO es eficiente: los cálculos anteriores son muy extensos y la sola enumeración de todos los programas es impráctica.*

4.6.5 Malas noticias: No hay detección universal de virus

Haciendo consideraciones más reales, tenemos que una computadora ejecuta a todo programa viéndolo como un subprograma de su propio sistema operativo. Un virus "real" "infecta al sistema operativo".

Refinemos los conceptos virales vistos hasta ahora. Consideremos un sistema operativo fijo, digamos SO.

- Un *virus* es un programa que altera el código del sistema operativo de una computadora.
- Un programa P *esparce* un virus bajo la entrada $\mathbf{x}$ si en el transcurso del cálculo de $P(\mathbf{x})$ se altera al SO.

- P es *seguro bajo la entrada* $\mathbf{x}$ si P no esparce un virus bajo $\mathbf{x}$.
- P es *seguro* si lo es bajo toda entrada $\mathbf{x}$.

Un programa *antiviral universal* es un programa tal que

$$AVU(P, \mathbf{x}) = \begin{cases} \textit{Pásale} & \text{si } P \text{ es seguro bajo } \mathbf{x}, \\ \textit{¡Alto!} & \text{en otro caso.} \end{cases}$$

Proposición 4.6.1 *No puede existir* AVU.

Demostración: Supongamos que se tuviera un programa antiviral universal *AVU*. Construyamos el programa D tal que

$$D(P) = \begin{cases} \textbf{Escribe } \textit{“No corra riesgos. Trabaje a lo seguro.”} & \text{si } AVU(P,P) = \textit{¡Alto!}, \\ \text{Modifica a SO} & \text{en otro caso.} \end{cases}$$

Tenemos pues que $\forall P$:

$$\begin{aligned} D \text{ es seguro bajo } P &\Leftrightarrow AVU(P,P) = \textit{¡Alto!} \\ &\Leftrightarrow P \text{ no es seguro bajo } P \end{aligned}$$

En particular, para $P = D$ se tiene una contradicción.

Capítulo 5

Irresolubilidad

En la sección 4.2 vimos que no existe una función recursiva que permita decidir, para una función recursiva f cualquiera y una instancia $\mathbf{x}$ arbitraria, si acaso $\mathbf{x}$ está o no en el dominio de f.

Dada la equivalencia entre funciones recursivas y máquinas de Turing, tenemos que el resultado de "inexistencia" anterior se convierte en que no hay máquina de Turing que decida, para una máquina de Turing cualquiera si acaso ha de detenerse a partir de una instancia inicial cualquiera. El "problema de la parada" para máquinas de Turing es pues *irresoluble* mediante máquinas de Turing. En resumen, existen problemas irresolubles que se pueden plantear en el lenguaje de la teoría de la computabilidad.

En el presente capítulo presentaremos diversos problemas que, al igual que el problema de la parada, no son resolubles mediante funciones computables.

Inicialmente, presentamos el teorema de Rice que indica que no hay manera alguna de reconocer procedimentalmente clases "interesantes" de funciones. Veremos luego otro problema clásico irresoluble, el de la "correspondencia de Post". A partir de éste, veremos que muchos problemas de decisión, planteados para gramáticas libres de contexto, vale decir, para gramáticas en los primeros niveles de la Jerarquía de Chomsky, son también irresolubles.

Finalmente, presentamos el equivalente a la noción de irresolubilidad en la Lógica Matemática. En ésta, una *demostración* consiste, al igual que una *computación*, de una secuencia de transformaciones para derivar *teoremas*. Un enunciado *indemostrable* es entonces uno inaccesible mediante las reglas de deducción de la teoría. Veremos un ejemplo de un enunciado indemostrable en la Aritmética de Peano.

5.1 Teorema de Rice

Consideremos la clase de funciones recursivas con un solo argumento. Sea pues $FRec^1 = \{f_k\}_{k\geq 0}$ una enumeración efectiva de las funciones computables de una sola variable.

Sea $\mathcal{F} \subset FRec^1$ una subclase. El conjunto de *índices* de $\mathcal{F}$ es $R_{\mathcal{F}} = \{i \geq 0 | f_i \in \mathcal{F}\}$. La clase $\mathcal{F}$ se dice ser *recursiva* si su conjunto de índices $R_{\mathcal{F}}$ lo es, es decir, si la función característica de este último conjunto es recursiva.

Disgresión: Supongamos que f es una función computable, calculada por el programa P y que el índice de f es k_0. Sea $\mathcal{F} = \{f\}$ la mónada que consta únicamente de la función f. Tenemos que $\{k_0\} \subset R_{\mathcal{F}}$. Pero la inclusión es estricta, pues f puede ser calculada también por una infinidad de otros programas, es decir, f tiene una infinidad de índices. Para determinar que un número cualquiera k está en $R_{\mathcal{F}}$ será menester que la función f_k coincide con f, la verificación de lo cual es un problema irresoluble, en general, como lo veremos más adelante. La dificultad principal en revisar si acaso una función coincide con otra, estriba en que aún cuando podamos probar uno a uno, y, potencialmente todos, los valores de ambas funciones, ese proceso es

interminable pues siempre quedarán más valores pendientes de revisar que los ya revisados.

Teorema 5.1.1 (Rice) *Ninguna subclase propia no vacía de funciones recursivas $\mathcal{F} \subset FRec^1$ puede ser recursiva. En otras palabras, si existen $f, g \in FRec^1$ tales que $f \in \mathcal{F}$ y $g \notin \mathcal{F}$ entonces $\mathcal{F}$ no puede ser recursiva.*

Demostración: Supongamos que $\mathcal{F}$ sea recursiva y que existen $f, g \in FRec^1$ tales que $f \in \mathcal{F}$ y $g \notin \mathcal{F}$.

Existe pues una función recursiva $P_{\mathcal{F}} \in FRec^1$ que coincide con la función característica de $R_{\mathcal{F}}$, es decir, tal que

$$P_{\mathcal{F}}(i) = \begin{cases} 1 & \text{si } i \in R_{\mathcal{F}}, \\ 0 & \text{en otro caso.} \end{cases}$$

Definamos $h : \mathbb{N}^2 \to \mathbb{N}$ de manera que

$$h(i,x) = \begin{cases} g(x) & \text{si } i \in R_{\mathcal{F}}, \\ f(x) & \text{en otro caso.} \end{cases}$$

h es recursiva pues se está definiendo por casos utilizando funciones recursivas. Por el Teorema de Parametrización, existe una función recursiva $s : N \to N$ tal que $\forall i, x : \ f_{s(i)}(x) = h(i,x)$. Por el Teorema de Recursión ha de existir un k_0 tal que $f_{k_0} = f_{s(k_0)}$. Así pues, $\forall x : \ f_{k_0}(x) = h(k_0, x)$, vale decir,

$$\forall x : \ f_{k_0}(x) = \begin{cases} g(x) & \text{si } f_{k_0} \in \mathcal{F}, \\ f(x) & \text{en otro caso.} \end{cases}$$

Ya que $f \in \mathcal{F}$ pero $g \notin \mathcal{F}$ tenemos las equivalencias siguientes:

$$k_0 \in R_{\mathcal{F}} \quad \Leftrightarrow \quad \forall x : f_{k_0}(x) = g(x) \quad \Leftrightarrow \quad f_{k_0} \notin \mathcal{F} \quad \Leftrightarrow \quad k_0 \notin R_{\mathcal{F}}$$

y esto es una contradicción.

Digamos que una clase de funciones es *trivial* si bien es vacía, o bien coincide con toda la clase $FRec^1$. El Teorema de Rice asevera pues que las únicas clases de funciones que son recursivas son precisamente las triviales.

Corolario 5.1.1 *Las siguientes subclases de $FRec^1$ NO son en sí recursivas, es decir, no hay programa alguno para reconocer cuándo una función dada en $FRec^1$ cae en una de esas clases.*

RP	$=$	$\{f \mid f$ es recursiva primitiva.$\}$	;	Ac	$=$	$\{f \mid f$ es acotada.$\}$
Fi	$=$	$\{f \mid f$ tiene una imagen finita.$\}$	;	To	$=$	$\{f \mid f$ es total.$\}$
$Df(x_0)$	$=$	$\{f \mid f(x_0)\downarrow\}$	;	Va	$=$	$\{f \mid \forall x : f(x)\uparrow\}$
$VaCTP$	$=$	$\{f \mid f$ diverge casi en todas partes.$\}$	;	$Eq(F)$	$=$	$\{f \mid f = F\}$
$Im(a)$	$=$	$\{f \mid \exists x : f(x) = a\}$	;	So	$=$	$\{f \mid f$ es suprayectiva.$\}$
In	$=$	$\{f \mid f$ es inyectiva.$\}$	;	Bi	$=$	$\{f \mid f$ es biyectiva.$\}$

El corolario se sigue de inmediato del teorema de Rice pues cada una de las subclases enlistadas es un subconjunto propio no-vacío de $FRec^1$.

Teorema 5.1.2 (Rice: Versión vectorial) *Sea $\mathcal{F}^m \subset \left(FRec^1\right)^m$ una familia de m-tuplas cuyas componentes son funciones recursivas. Sea $R_{\mathcal{F}^m} = \{(i_1, \ldots, i_m) | (f_{i_1}, \ldots, f_{i_m}) \in \mathcal{F}^m\}$.*

Entonces $R_{\mathcal{F}^m}$ no puede ser recursivo a menos de que $R_{\mathcal{F}^m}$ sea trivial, es decir, $R_{\mathcal{F}^m} = \emptyset$ o $R_{\mathcal{F}^m} = \mathbb{N}^m$

Demostración: Sea $a_m : \mathbb{N}^m \to \mathbb{N}$ una biyección computable. Dado $\mathcal{F}^m$, definimos

$$S = \{i \in \mathbb{N} | \exists \mathbf{f} = (f_{i_1}, \ldots, f_{i_m}) \in \mathcal{F}^m : i = a_m(i_1, \ldots, i_m)\}.$$

Se tiene que $R_{\mathcal{F}^m}$ es recursivo si y sólo si S lo es, y S no es trivial si y sólo si $R_{\mathcal{F}^m}$ no lo es. Del Teorema de Rice se sigue que S no es recursivo.

Corolario 5.1.2 *Las siguientes clases de funciones NO son recursivas.*

$$\begin{array}{lcl;{5pt/5pt}lcl} Ig & = & \{(f,g)|f=g\} & ; & MeI & = & \{(f,g)|\forall x:(f(x)\downarrow \wedge g(x)\downarrow \Rightarrow f(x)\leq g(x)\} \\ Cp & = & \{(f,g,h)|h=g\circ f\} & ; & OB & = & \{(f,g,h)|h=g\oplus f\} \end{array}$$

donde $\oplus$ es, por ejemplo, una operación binaria computable.

El corolario se sigue de que las clases enlistadas no son triviales.

Se dice que una función g es una *extensión* de la función f, o que f es una *restricción* de g, si se cumple

$$\forall x:\ f(x)\downarrow \Rightarrow\ [g(x)\downarrow\ \wedge\ [g(x)=f(x)]]\,.$$

g extiende propiamente a f si existe x_0 tal que $g(x_0)\downarrow$ pero $f(x_0)\uparrow$.

Teorema 5.1.3 (Primera extensión del Teorema de Rice) *Sea $\mathcal{F}\subset FRec^1$ una familia de funciones recursivas tal que existe una función $f\in\mathcal{F}$ con una extensión propia recursiva g que NO está en $\mathcal{F}$.*

Entonces el conjunto de índices $R_{\mathcal{F}}=\{i\geq 0|f_i\in\mathcal{F}\}$ no puede ser recursivamente enumerable.

Demostración: Sean f y g como en el enunciado del teorema. Sea $(f_i)_i$ una enumeración efectiva de las funciones recursivas. Consideremos la función $h:\mathbb{N}^2\to\mathbb{N}$ tal que

$$h(i,x)=\begin{cases} f(x) & \text{si } f_i(i)\uparrow, \\ g(x) & \text{en otro caso.}\end{cases}$$

La función $i\mapsto f_i(i)$ es claramente computable, pues es la diagonal de una función universal. Sea P un programa que la calcule. Como f y g son recursivas, existen sendos programas Q_1,Q_2 que las calculan. La función h puede calcularse como sigue:

Dado (i,n),

1. calculamos $P(i)$ y $Q_1(x)$ en "tiempo compartido", como si se calculara $P(i)\wedge Q_1(x)$ según el procedimiento descrito en 3.4.6,
2. si el cálculo de $Q_1(x)$ termina primero, se da su valor como salida,
3. si el cálculo de $P(i)$ termina primero, se pasa a calcular $Q_2(x)$ y se da su valor como salida.

Puesto que g es una extensión de f el procedimiento anterior calcula $h(i,x)$. Por tanto, h es recursiva.

Por el Teorema de Parametrización existe una función computable s tal que $\forall i,x: f_{s(i)}(x)=h(i,x)$. Así pues, $\forall i:[s(i)\in R_{\mathcal{F}}\Leftrightarrow f_i(i)\uparrow]$. Ahora bien, el conjunto $K=\{i|f_i(i)\downarrow\}$ bien que es recursivamente enumerable no es recursivo (éste aparece de manera esencial en el Problema de la Parada). Se tiene además que $\forall i:[s(i)\in R_{\mathcal{F}}\Leftrightarrow i\notin K]$. Consecuentemente, si $R_{\mathcal{F}}$ fuera recursivamente enumerable entonces también lo sería el complemento de K y esto obligaría a que K fuera recursivo, lo cual no es posible.

Corolario 5.1.3 *Las siguientes clases de funciones NO son recursivamente enumerables, es decir, no hay programa alguno para generar uno a uno a todos los elementos en una de esas clases.*

$$\begin{array}{lcl;{5pt/5pt}lcl} To^c & = & \{f|f \text{ no es total.}\} & ; & Eq^c(F) & = & \{f|f\neq F\}\text{, con } F \text{ total,} \\ Eq(G) & = & \{f|f=G\}\text{, con } G \text{ no-total,} & ; & Df^c(x_0) & = & \{f|f(x_0)\uparrow\} \\ Va & = & \{f|\forall x: f(x)\uparrow\} & ; & VaCTP & = & \{f|f \text{ diverge casi en todas partes.}\} \\ Im^c(a) & = & \{f|\forall x: f(x)\neq a\} & ; & Fi & = & \{f|f \text{ tiene una imagen finita.}\} \\ So^c & = & \{f|f \text{ no es suprayectiva.}\} & ; & In & = & \{f|f \text{ es inyectiva.}\} \\ Bi^c & = & \{f|f \text{ no es biyectiva.}\} & & & & \end{array}$$

En efecto, para cada una de esas clases se puede dar un ejemplo particular de una función f en la clase con una extensión fuera de ella.

5.2 Problema de la correspondencia de Post

Hemos visto que el problema de la parada en máquinas de Turing no es resoluble por máquinas de Turing.

Así pues, con la terminología y la notación introducidas en la sección 1.6.2, tenemos que no es posible construir un procedimiento recursivo que decida si acaso una *descripción instantánea inicial*, DI_0, ha de transformarse en una DI final.

5.2.1 Presentación del Problema de Post

Sea A un alfabeto finito y sea A^* su diccionario. El *problema de la correspondencia de Post* (PCP) se define como sigue:

Instancia: Dos listas $\mathbf{X} = [\mathbf{x}_i]_{i=1,\ldots,n}$ e $\mathbf{Y} = [\mathbf{y}_i]_{i=1,\ldots,n}$, con un mismo número de elementos, formadas por palabras en A^*.

Solución: Una lista de índices $\mathbf{I} \in \{1,\ldots,n\}^{\mathrm{long}(\mathbf{I})}$ tal que las yuxtaposiciones de los elementos de $\mathbf{X}$ e $\mathbf{Y}$, de acuerdo con la lista $\mathbf{I}$ coinciden, i.e.

$$\mathbf{I} = [i_j]_{j=1,\ldots,m} \;\Rightarrow\; [\mathbf{x}_{i_1} \cdots \mathbf{x}_{i_m}] = [\mathbf{y}_{i_1} \cdots \mathbf{y}_{i_m}].$$

Ejemplos

Consideremos

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \mathbf{x}_4 \\ \mathbf{x}_5 \\ \mathbf{x}_6 \\ \mathbf{x}_7 \end{bmatrix} = \begin{bmatrix} 100 \\ 10 \\ 11101 \\ 01 \\ 101 \\ 11110 \\ 01101 \end{bmatrix} \quad ; \quad \mathbf{Y} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \mathbf{y}_3 \\ \mathbf{y}_4 \\ \mathbf{y}_5 \\ \mathbf{y}_6 \\ \mathbf{y}_7 \end{bmatrix} = \begin{bmatrix} 101 \\ 01101 \\ 11011 \\ 0 \\ 1 \\ 1110 \\ 11101 \end{bmatrix}$$

Para $\mathbf{I} = [5, 2, 3, 3, 6]$ se tiene

$$\mathbf{x}_5\mathbf{x}_2\mathbf{x}_3\mathbf{x}_3\mathbf{x}_6 = \mathbf{y}_5\mathbf{y}_2\mathbf{y}_3\mathbf{y}_3\mathbf{y}_6$$

pues

$$\begin{aligned} 10110111011110111110 &= 101.10.11101.11101.11110 \\ &= 1.01101.11011.11011.1110 \end{aligned}$$

En cambio para

$$\begin{aligned} \mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3] &= [11 \quad 100 \quad 0] \\ \mathbf{Y} = [\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3] &= [00 \quad 1 \quad 101] \end{aligned}$$

el PCP no tiene solución alguna. (¡Revísese!)

El *problema de la correspondencia de Post modificado* (PCPM) coincide con el anterior salvo en que en este último se impone la exigencia de que la cadena común formada comience con las primeras palabras en las listas dadas:

Instancia: Dos listas $\mathbf{X} = [\mathbf{x}_i]_{i=1,\ldots,n}$ e $\mathbf{Y} = [\mathbf{y}_i]_{i=1,\ldots,n}$, con un mismo número de elementos, formadas por palabras en A^*.

Solución: Una lista de índices $\mathbf{I} \in \{1,\ldots,n\}^{\mathrm{long}(\mathbf{I})}$ tal que

$$\mathbf{I} = [i_j]_{j=1,\ldots,m} \;\Rightarrow\; [\mathbf{x}_1\mathbf{x}_{i_1} \cdots \mathbf{x}_{i_m}] = [\mathbf{y}_1\mathbf{y}_{i_1} \cdots \mathbf{y}_{i_m}].$$

Proposición 5.2.1 *Si PCP fuera efectivamente resoluble entonces PCPM lo sería también.*

Demostración: Traduciremos toda instancia $(\mathbf{X}^M, \mathbf{Y}^M)$ de PCPM, con n palabras en cada lista, a una instancia $(\mathbf{X}, \mathbf{Y})$ de PCP de manera que la instancia traducida posea una solución cuando y sólo cuando la instancia original también posea una correspondiente solución. Al traducir utilizaremos dos nuevos símbolos *separadores* "$\circ, \bullet$".

Ilustraremos el procedimiento con el ejemplo anterior con solución $\mathbf{I} = [5, 2, 3, 3, 6]$. Para partir de una solución de PCPM, intercambiaremos las palabras primera y quinta en cada conjunto de palabras:

$$\mathbf{X}^M = \begin{bmatrix} 101 \\ 10 \\ 11101 \\ 01 \\ 100 \\ 11110 \\ 01101 \end{bmatrix} \quad ; \quad \mathbf{Y}^M = \begin{bmatrix} 1 \\ 01101 \\ 11011 \\ 0 \\ 101 \\ 1110 \\ 11101 \end{bmatrix}$$

La solución es pues $\mathbf{I} = [1, 2, 3, 3, 6]$.

Para hacer la conversión, cambiaremos primero a las listas de palabras $\mathbf{X}^M, \mathbf{Y}^M$, después introduciremos nuevas palabras iniciales y nuevas palabras terminales.

- *Construcción de* $\mathbf{X}$: $\mathbf{X}$ se obtiene de $\mathbf{X}^M$ sustituyendo cada símbolo a de A por $a\circ$.

$$\mathbf{X} = \begin{bmatrix} 1\circ 0\circ 1\circ \\ 1\circ 0\circ \\ 1\circ 1\circ 1\circ 0\circ 1\circ \\ 0\circ 1\circ \\ 1\circ 0\circ 0\circ \\ 1\circ 1\circ 1\circ 1\circ 0\circ \\ 0\circ 1\circ 1\circ 0\circ 1\circ \end{bmatrix}$$

- *Construcción de* $\mathbf{Y}$: $\mathbf{Y}$ se obtiene de $\mathbf{Y}^M$ sustituyendo cada símbolo a de A por $\circ a$.

$$\mathbf{Y} = \begin{bmatrix} \circ 1 \\ \circ 0\circ 1\circ 1\circ 0\circ 1 \\ \circ 1\circ 1\circ 0\circ 1\circ 1 \\ \circ 0 \\ \circ 1\circ 0\circ 1 \\ \circ 1\circ 1\circ 1\circ 0 \\ \circ 1\circ 1\circ 1\circ 0\circ 1 \end{bmatrix}$$

- *Construcción de palabras iniciales*: Hagamos $\mathbf{x}_0 = \circ\mathbf{x}_1$ e $\mathbf{y}_0 = \mathbf{y}_1$.

$$\begin{aligned} \mathbf{x}_0 &= \circ 1\circ 0\circ 1\circ \\ \mathbf{y}_0 &= \circ 1 \end{aligned}$$

- *Construcción de palabras terminales*: Hagamos $\mathbf{x}_{n+1} = \bullet$ e $\mathbf{y}_{n+1} = \circ\bullet$.

$$\begin{aligned} \mathbf{x}_8 &= \bullet \\ \mathbf{y}_8 &= \circ\bullet \end{aligned}$$

Veamos ahora cómo corresponden soluciones de PCPM a soluciones de PCP y viceversa.

	Lista $\mathbf{X}$	*Lista* $\mathbf{Y}$	*Explicación*
	$\bullet$	$\bullet q_0 \mathbf{x} \bullet$	Inscríbase la DI inicial correspondiente a $\mathbf{x}$.
$\forall a \in A$	a	a	Déjese sin cambio alguno a los símbolos de A.
	$\bullet$	$\bullet$	Déjese sin cambio alguno al símbolo terminal $\bullet$.

Tabla 5.1: Grupo I: Parejas iniciales y de símbolos en la m. T.

- *Una solución de* $(\mathbf{X}^M, \mathbf{Y}^M)$ *en PCPM da una solución de* $(\mathbf{X}, \mathbf{Y})$ *en PCP*: Si $\mathbf{I}^M = [1, i_1, \ldots, i_k]$ es una solución de PCPM entonces $\mathbf{I} = [0, i_1, \ldots, i_k, n+1]$ es solución de PCP.

  En el ejemplo mostrado, la palabra común dada por la solución $\mathbf{I} = [0, 2, 3, 3, 6, 8]$, inducida a su vez por la solución $\mathbf{I}^M = [1, 2, 3, 3, 6]$ es

  $$\circ 1 \circ 0 \circ 1 \circ 1 \circ 0 \circ 1 \circ 1 \circ 1 \circ 0 \circ 1 \circ 1 \circ 1 \circ 0 \circ 1 \circ 1 \circ 1 \circ 1 \circ 1 \circ 0 \circ \bullet$$

- *Una solución de* $(\mathbf{X}, \mathbf{Y})$ *en PCP da una solución de* $(\mathbf{X}^M, \mathbf{Y}^M)$ *en PCPM*: Si $\mathbf{I} = [i_1, i_2 \ldots, i_{k-1}, i_k]$ es una solución de PCP entonces necesariamente $i_1 = 0$ e $i_k = n+1$, pues ésta es la única manera de que "empaten" las marcas extremas $\circ, \bullet$; es decir, la palabra común ha de comenzar con las palabras iniciales y ha de terminar con las terminales. Sea i_j el mínimo índice tal que $i_j = n+1$. Evidentemente $\mathbf{I}' = [i_1, i_2 \ldots, i_{j-1}, i_j]$ también es solución de PCP. Se ve de manera directa que $\mathbf{I}^M = [1, i_2, \ldots, i_{j-1}]$ es solución de PCPM.

Teorema 5.2.1 *PCP es irresoluble.*

Demostración: Veremos que PCPM no puede resolverse efectivamente. Para esto veremos cómo una instancia del Problema de la Parada en máquinas de Turing (PPmT) se reduce algorítmicamente a una instancia de PCPM de manera que la instancia a reducirse ha de poseer una solución, es decir, se reconocerá que la máquina "instancia" se parará a partir del contenido inicial "instancia", en PPmT si y sólo si la correspondiente instancia ya reducida posee una solución para PCPM. Siendo entonces PPmT reducible a PCPM se sigue inmediatamente que este último problema no puede ser resoluble pues si lo fuera, PCmT también lo sería, lo cual es imposible.

Sea pues $(M, \mathbf{w})$ una instancia de PPmT: M es una mT y $\mathbf{w}$ es una entrada para M. Sea $AQ = A \cup Q \cup \{\bullet\}$ el alfabeto consistente de los símbolos y los estados de M, y de un símbolo para indicar extremos de DI's. Construiremos dos sucesiones de palabras $\mathbf{X}, \mathbf{Y} \subset AQ^*$ tales que $(\mathbf{X}, \mathbf{Y})$ posee solución en PCP si y sólo si $M(\mathbf{w}) \downarrow$ y en tal caso una solución de PCPM ha de consistir, propiamente, de una computación terminal que transforma la DI inicial $q_0\mathbf{x}$ en alguna DI final.

Las construcciones de ambas listas $\mathbf{X}, \mathbf{Y}$ se hacen por etapas. En las tablas 5.1-5.3 quedan mostradas las palabras formadas para sendas listas $\mathbf{X}, \mathbf{Y}$ divididas por grupos: en el primer grupo presentamos partículas iniciales y las relativas al alfabeto de la m.T., en el segundo grupo aparecen las que codifican el funcionamiento de la m.T. y en el tercer grupo se enlista a las partículas terminales.

Veamos que una solución de PPmT da una solución para PCPM.

Se dice que una pareja $(C, D) \in (AQ^*)^2$ es una *solución parcial* de PCPM si

- C es un prefijo de D, y
- C y D son yuxtaposiciones de correspondientes cadenas en las listas $\mathbf{X}$ y $\mathbf{Y}$, es decir, si $C = \mathbf{x}_{i_1} \ldots \mathbf{x}_{i_k}$ entonces $D = \mathbf{y}_{i_1} \ldots \mathbf{y}_{i_k} E$, donde $E \in AQ^*$. En tal caso, la partícula E es el *residuo* de la solución parcial (C, D).

Se ve que las siguientes dos proposiciones son equivalentes:

	Lista **X**	*Lista* **Y**	*Explicación*
$(q, a, p, b, Der) \in t$:	qa	bp	Aplíquese quintuplas cuyos movimientos son a la "derecha".
$(q, a, p, b, Izq) \in t$:	$a'qa$	$pa'b$	Aplíquese quintuplas cuyos movimientos son a la "izquierda".
$(q, a_0, p, b, Der) \in t$:	$q\bullet$	$bp\bullet$	Aplíquese quintuplas, correspondientes al símbolo "blanco", en el extremo derecho de la cinta, cuyos movimientos son a la "derecha".
$(q, a_0, p, b, Izq) \in t$:	$a'q\bullet$	$pa'b\bullet$	Aplíquese quintuplas, correspondientes al símbolo "blanco", en el extremo derecho de la cinta, cuyos movimientos son a la "izquierda".

Tabla 5.2: Grupo II: Parejas de transiciones.

	Lista **X**	*Lista* **Y**	*Explicación*
$q \in F, a, a' \in A$	aqa'	q	Al arribar a un estado final comiéncese a suprimir símbolos para obtener una DI vacía.
$q \in F, a \in A$	aq	q	Idem.
$q \in F, a' \in A$	qa'	q	Idem.
	$q \bullet \bullet$	$\bullet$	Al obtener la DI vacía, deténgase el proceso.

Tabla 5.3: Grupo III: Parejas de estados finales y DI's terminales.

1. La sucesión de DI's $\mathbf{DI} = \{DI_i = \boldsymbol{\alpha}_i q_i a_i \boldsymbol{\beta}_i\}_{i=0,\ldots,k}$ es una computación, intermedia o terminal, de longitud $k+1$ en la m. T. M, es decir $\forall i < k : M \vdash DI_i \to DI_{i+1}$.

2. Si

$$\begin{aligned} C_k &= q_0\mathbf{w} \bullet \boldsymbol{\alpha}_1 q_1 a_1 \boldsymbol{\beta}_1 \bullet \cdots \bullet \boldsymbol{\alpha}_{k-1} q_{k-1} a_{k-1} \boldsymbol{\beta}_{k-1} \qquad \text{y} \\ D_k &= q_0\mathbf{w} \bullet \boldsymbol{\alpha}_1 q_1 a_1 \boldsymbol{\beta}_1 \bullet \cdots \bullet \boldsymbol{\alpha}_{k-1} q_{k-1} a_{k-1} \boldsymbol{\beta}_{k-1} \bullet \boldsymbol{\alpha}_k q_k a_k \boldsymbol{\beta}_k E_k \end{aligned}$$

   entonces (C_k, D_k) es una solución parcial de PCPM.

Las dos implicaciones a demostrar se prueban por inducción en k.

Ahora, con la notación anterior, observamos que la solución parcial de la forma (C_k, D_k) más pequeña corresponde a cuando E_k es la palabra vacía. También observamos que si no aparecen palabras del grupo III, entonces en toda solución parcial (C_k, D_k), la longitud de D_k es estrictamente mayor que la de C_k. Por tanto, se tendrá una solución a PCPM si y sólo si se tiene una computación terminal en la máquina dada.

5.2.2 Una aplicación de PCP

Veremos aquí un problema típico relativo a gramáticas libres de contexto que es irresoluble, y cuya irresolubilidad se demuestra viendo que es equivalente al Problema de la Correspondencia de Post.

Recordamos las siguientes definiciones

- Una palabra w en el lenguaje libre de contexto (LLC) L es *ambigua* en una GLC G que genere a L si w posee más de una derivación *siniestra (leftmost)* en G.

  Como un ejemplo trivial, se tiene que, en la gramática con producciones $S \to A|B$, $A \to a$, $B \to a$, la palabra a posee dos derivaciones siniestras

$$S \to A \to a \quad ; \quad S \to B \to a$$

- Una GLC G es *ambigua* si hay una palabra, del lenguaje que genera, ambigua en ella.
- Un LLC L es *inherentemente ambiguo* si cualquier GLC G que lo genere es ambigua.

  A guisa de ejemplo se tiene que el lenguaje

  $$L = \{a^n b^n c^m d^m | n, m \geq 1\} \cup \{a^n b^m c^m d^n | n, m \geq 1\} = L_1 \cup L_2.$$

  es inherentemente ambiguo.

  Sin que presentemos aquí una demostración formal de esta aseveración, daremos una motivaci"on intuitiva de ella. Seguramente el lector no tendrá ninguna dificultad en visualizar una GLC para generar L_1 y otra para generar L_2. Ahora, consideremos el lenguaje $L_0 = \{a^n b^n c^n d^n | n \geq 1\}$. Tenemos que $L_0 \subset L_1 \cap L_2 \subset L$ y cada palabra en L_0 puede ser derivada tanto por la gramaática de L_1 como por la gramaática de L_2. Una y otra derivación hacen que cada papalabra de L_0 se a ambigua (para cualquier gramática de L).

El *Problema de Reconocimiento de Ambigüedad en GLC (PRAGLC)* es el siguiente:

Instancia: Una gramática libre de contexto G.

Solución: $\begin{cases} 1 & \text{si } G \text{ es ambigua,} \\ 0 & \text{en otro caso.} \end{cases}$

Proposición 5.2.2 *PCP se reduce algorítmicamente a PRAGLC. Por tanto este último es irresoluble.*

Demostración: Dada una instancia $(\mathbf{X}, \mathbf{Y})$ de PCP hemos de construir una instancia I_{PRAGLC} de PRAGLC, la cual consistirá de una gramática $G_{(\mathbf{X},\mathbf{Y})}$, de manera que $(\mathbf{X}, \mathbf{Y})$ posea solución en PCP si y sólo si $G_{(\mathbf{X},\mathbf{Y})}$ sea ambigua.

Denotemos por A al alfabeto de PCP. Escribamos $\mathbf{X} = [\mathbf{x}_1 \cdots \mathbf{x}_k]$, $\mathbf{Y} = [\mathbf{y}_1 \cdots \mathbf{y}_k]$, y extendamos A incluyendo k nuevos símbolos $b_1, \ldots, b_k$. Construyamos los siguientes dos lenguajes:

$$\begin{aligned} L_{\mathbf{X}} &= \{\mathbf{x}_{i_1} \cdots \mathbf{x}_{i_m} b_{i_m} \cdots b_{i_1} | [i_1, \ldots, i_m] \in [1,k]^m, m \geq 1\} \\ L_{\mathbf{Y}} &= \left\{\mathbf{y}_{i_1} \cdots \mathbf{y}_{i_m} b_{i_m} \cdots b_{i_1} | [i_1, \ldots, i_m] \in [1,k]^m, m \geq 1\right\}. \end{aligned}$$

Sea $G_{(\mathbf{X},\mathbf{Y})} = (\{S, S_{\mathbf{X}}, S_{\mathbf{Y}}\}, A \cup \{b_i\}_{i \leq k}, P, S)$ la GLC cuyas producciones son:

$$\begin{aligned} S &\rightarrow S_{\mathbf{X}} | S_{\mathbf{Y}} \\ \forall i \leq k: \ S_{\mathbf{X}} &\rightarrow \mathbf{x}_i S_{\mathbf{X}} b_i | \mathbf{x}_i b_i \\ \forall i \leq k: \ S_{\mathbf{Y}} &\rightarrow \mathbf{y}_i S_{\mathbf{Y}} b_i | \mathbf{y}_i b_i \end{aligned}$$

Resultan evidentes las propiedades siguientes:

- El lenguaje generado por esta gramática es $L(G_{(\mathbf{X},\mathbf{Y})}) = L_{\mathbf{X}} \cup L_{\mathbf{Y}}$.
- $G_{(\mathbf{X},\mathbf{Y})}$ es ambigua si y sólo si se cumple cualquiera de las siguientes dos proposiciones claramente equivalentes entre sí:
  1. $L_{\mathbf{X}} \cap L_{\mathbf{Y}} \neq \emptyset$
  2. $(\mathbf{X}, \mathbf{Y})$ posee solución en PCP.

5.3 Irresolubilidad de problemas en GLC

5.3.1 Reducción de problemas al problema de la parada

Como una primera sección, veamos que las computaciones terminales en una máquina de Turing dada, pueden "realizarse" como la intersección de dos lenguajes libres de contexto. En otras palabras, el proceso de "correr una máquina de Turing sobre una entrada dada" puede realizarse por dos autómatas de pila "corriendo en paralelo".

Sea $M = (Q, A, t, q_0, F)$ una máquina de Turing. Sea **DI** una computación, terminal o intermedia. Supongamos $\mathbf{DI} = \{DI_i\}_{i=0,\ldots,k} = \{\mathbf{y}_i q_i a_i \mathbf{x}_i\}_{i=0,\ldots,k}$. La *representación lineal* de **DI** es la palabra

$$DI_0 \bullet DI_1 \bullet \cdots \bullet DI_{k-1} \bullet DI_k,$$

y la *representación trastocada* de **DI** es la palabra

$$DI_0 \bullet DI_1^{\text{rev}} \bullet DI_2 \bullet DI_3^{\text{rev}} \cdots,$$

consistente de la misma palabra que en la representación lineal, salvo que las descripciones correspondientes a índices impares se escriben al revés.

Sea $L_{\rightarrow} = \{DI_0 \bullet DI_1^{\text{rev}} | DI_0, DI_1 \text{ son DI's de } M \text{ y } M \vdash [DI_0 \rightarrow DI_1]\}$ el lenguaje consistente de "inicios" de computaciones legales en la máquina de Turing.

Proposición 5.3.1 *$L_{\rightarrow}$ es un lenguaje libre de contexto.*

Demostración: $L_{\rightarrow}$ es reconocido por un autómata de pila cuyo comportamiento es el siguiente:

Dada una palabra de la forma $\mathbf{d}_0 \bullet \mathbf{d}_1$,

1. lee uno a uno los símbolos de $\mathbf{d}_0$, hasta llegar a "$\bullet$", y verifica que aparezca exactamente una vez un símbolo correspondiente a un estado, es decir, reconoce a $\mathbf{d}_0$ de la forma $\mathbf{d}_0 = \mathbf{y}_0 q_0 \mathbf{x}_0$,
2. reconoce a la "parte izquierda" de $\mathbf{d}_1$ como de la forma $\mathbf{x}_0^{\text{rev}}$, acaso tal vez difiriendo en una sola posición,
3. verifica que donde termina $\mathbf{x}_0^{\text{rev}}$ aparezca la aplicación de una transición de la máquina M, y, finalmente
4. reconoce a la "parte derecha" de $\mathbf{d}_1$ como de la forma $\mathbf{y}_0^{\text{rev}}$ acaso tal vez difiriendo en una sola posición.

Es claro que $L_{\rightarrow}^{\text{rev}}$ es también libre de contexto. Se tiene

$$L_{\rightarrow}^{\text{rev}} = \{DI_0^{\text{rev}} \bullet DI_1 | DI_0, DI_1 \text{ son DI's de } M \text{ y ahí } DI_0 \rightarrow DI_1\}.$$

Sea $L_{\underset{*}{\rightarrow}}^{F}(\mathbf{x}_0)$ el lenguaje consistente de representaciones trastocadas de computaciones terminales en M que se inician con la descripción inicial $q_0\mathbf{x}_0$.

Proposición 5.3.2 *$L_{\underset{*}{\rightarrow}}^{F}(\mathbf{x}_0)$ es la intersección de dos lenguajes libres de contexto.*

Demostración: Se comprueba inmediatamente que la proposición se cumple considerando los lenguajes, evidentemente libres de contexto,

$$\begin{array}{lcll} L_1 & = & & (L_{\rightarrow}\bullet)^* (\{nil\} \cup \text{DI\_final}\bullet) \\ L_2 & = & q_0\mathbf{x}_0\bullet & (L_{\rightarrow}^{\text{rev}}\bullet)^* (\{nil\} \cup \text{DI\_final}\bullet) \end{array}$$

Veamos a partir de este hecho que algunos problemas relativos a lenguajes libres de contexto son irresolubles.

Problema de *"gramáticas ajenas"* (PA)

Instancia: Dos gramáticas libres de contexto G_1, G_2 con un mismo alfabeto terminal A.

Solución: $\begin{cases} 1 & \text{si } G_1 \cap G_2 = \emptyset, \\ 0 & \text{en otro caso.} \end{cases}$

Proposición 5.3.3 *PA es irresoluble.*

Demostración: Si pudiéramos decidir cuándo dos lenguajes libres de contexto tienen intersección no-vacía, al considerar los lenguajes L_1 y L_2 de la proposición anterior podríamos decidir si la máquina M posee una computación terminal a partir de $\mathbf{x}_0$. Así pues resolveríamos el Problema de la Parada para máquinas de Turing. Como esto es imposible, PA es irresoluble.

Problema de *totalidad* (PT)

Instancia: Una gramática libre de contexto G sobre un alfabeto terminal A.

Solución: $\begin{cases} 1 & \text{si } L(G) = A^*, \\ 0 & \text{en otro caso.} \end{cases}$

Proposición 5.3.4 *PT es irresoluble.*

Demostración: Puede verse que el lenguaje complementario a las representaciones trastocadas de computaciones en la máquina de Turing es libre de contexto. En efecto, una palabra σ está en el lenguaje complementario si se cumple alguna de las condiciones siguientes:

1. No es de la forma $\mathbf{DI} = \left\{ (DI_i)^{(-1)^i} \right\}_{i=0,\ldots,k} = \left\{ (\mathbf{y}_i q_i a_i \mathbf{x}_i)^{(-1)^i} \right\}_{i=0,\ldots,k}$, donde $\tau^1 = \tau$ y $\tau^{-1} = \tau^{\mathrm{rev}}$.
2. Siendo de la forma **DI**, se tiene que la primera $\mathbf{y}_0 q_0 a_0 \mathbf{x}_0$ no es inicial (es decir, el q_0 que ahí aparece no es el estado inicial de la máquina de Turing).
3. Siendo de la forma **DI**, se tiene que la última $\mathbf{y}_k q_k a_k \mathbf{x}_k$ no es terminal, es decir, $q_k \notin F$.
4. Para alguna i par se tiene $\not\vdash \mathbf{y}_i q_i a_i \mathbf{x}_i \rightarrow \left(\mathbf{y}_{i+1} q_{i+1} a_{i+1} \mathbf{x}_{i+1}\right)^{\mathrm{rev}}$.
5. Para alguna i impar se tiene $\not\vdash (\mathbf{y}_i q_i a_i \mathbf{x}_i)^{\mathrm{rev}} \rightarrow \mathbf{y}_{i+1} q_{i+1} a_{i+1} \mathbf{x}_{i+1}$.

Cada una de las tres primeras condiciones define a un lenguaje regular, y por tanto, libre de contexto. La revisión de las condiciones 4. y 5. puede hacerse mediante sendos autómatas de pila y por tanto éstas determinan lenguajes libres de contexto. Así pues, el lenguaje complementario es libre de contexto como unión de lenguajes libres de contexto.

Si pudiéramos reconocer que este lenguaje es total podríamos reconocer que no hay computación terminal para una entrada dada de la máquina de Turing. Esto también resolvería el Problema de la Parada.

Problema de *equivalencia* (PE)

Instancia: Dos gramáticas libres de contexto G_1, G_2 con un mismo alfabeto terminal A.

Solución: $\begin{cases} 1 & \text{si } L(G_1) = L(G_2), \\ 0 & \text{en otro caso.} \end{cases}$

Proposición 5.3.5 *PE es irresoluble.*

Demostración: En particular si consideramos una GLC G_1 que genere a todo el alfabeto, si resolviéramos PE resolveríamos también PT.

La irresolubilidad de los siguientes problemas se debe a esta misma razón.

Problema de *inclusión* (PI)

Instancia: Dos gramáticas libres de contexto G_1, G_2 con un mismo alfabeto terminal A.

Solución: $\begin{cases} 1 & \text{si } L(G_1) \subset L(G_2), \\ 0 & \text{en otro caso.} \end{cases}$

Proposición 5.3.6 *PC es irresoluble.*

Problema de *regularidad* (PR)

Instancia: Una gramática libre de contexto G y un lenguaje regular R, ambos sobre un mismo alfabeto terminal A.

Solución: $\begin{cases} 1 & \text{si } R = L(G), \\ 0 & \text{en otro caso.} \end{cases}$

Proposición 5.3.7 *PR es irresoluble.*

Problema de *regularidad contenida* (PRC)

Instancia: Una gramática libre de contexto G y un lenguaje regular R, ambos sobre un mismo alfabeto terminal A.

Solución: $\begin{cases} 1 & \text{si } R \subset L(G), \\ 0 & \text{en otro caso.} \end{cases}$

Proposición 5.3.8 *PA es irresoluble.*

5.3.2 Algunos otros problemas irresolubles

Recordemos el

Lema de Bombeo para GLC's: *Sea L un LLC. Existe una constante $n \in \mathbb{N}$ tal que si $\mathbf{z} \in L$ es una palabra de longitud al menos n entonces la palabra $\mathbf{z}$ se puede descomponer como $\mathbf{z} = \mathbf{z}_1\mathbf{z}_2\mathbf{z}_3\mathbf{z}_4\mathbf{z}_5$, de manera que $\text{long}(\mathbf{z}_2) + \text{long}(\mathbf{z}_4) \geq 1$, $\text{long}(\mathbf{z}_2\mathbf{z}_3\mathbf{z}_4) \leq n$, y $\forall k \in \mathbb{N}: \ \mathbf{z}_1\mathbf{z}_2^k\mathbf{z}_3\mathbf{z}_4^k\mathbf{z}_5 \in L$.*

Ahora bien toda máquina de Turing es equivalente, mediante la adición de estados redundantes, a una máquina de Turing tal que toda computación terminal involucra al menos tres DI's.

De aquí se tiene el

Lema 5.3.1 *Sea M una máquina de Turing. Sea C_M el conjunto de computaciones terminales en M. Entonces*

$$C_M \text{ es un LLC} \Leftrightarrow L(M) \text{ es un lenguaje finito.}$$

Demostración: $\Rightarrow$) Si $L(M)$ fuera infinito y C_M libre de contexto se contradiría al Lema de Bombeo.

$\Leftarrow$) Es evidente, pues todo lenguaje finito es libre de contexto.

Problema de *complemento libre de contexto* (PCLC)

Instancia: Una gramática libre de contexto G sobre un alfabeto terminal A.

Solución: $\begin{cases} 1 & \text{si } A^* - L(G) \text{ es libre de contexto,} \\ 0 & \text{en otro caso.} \end{cases}$

Proposición 5.3.9 *PCLC es irresoluble.*

Demostración: Sea G la gramática libre de contexto que genera a las computaciones inválidas de M. Tenemos pues que $A^* - L(G)$ es libre de contexto si y sólo si el lenguaje de M es finito. Como esto último, de acuerdo con el Teorema de Rice, es indecidible, tenemos que PCLC es irresoluble.

Problema de *intersección libre de contexto* (PILC)

Instancia: Dos gramáticas libres de contexto G_1, G_2 sobre un mismo alfabeto terminal A.

Solución: $\begin{cases} 1 & \text{si } L(G_1) \cap L(G_2) \text{ es libre de contexto,} \\ 0 & \text{en otro caso.} \end{cases}$

Proposición 5.3.10 *PILC es irresoluble.*

Demostración: Sean G_1 y G_2 dos gramáticas libres de contexto tales que su intersección dé las computaciones terminales de M. Tenemos pues que $L(G_1) \cap L(G_2)$ es libre de contexto si y sólo si el lenguaje de M es finito. Como esto último, de acuerdo con el Teorema de Rice, es indecidible, tenemos que PILC es irresoluble.

5.4 Irresolubilidad en la Aritmética

5.4.1 Aritmética de Peano

Una *lógica* se construye sobre un *alfabeto*, el cual consiste de símbolos *especiales*, los cuales son, usualmente, los conectivos lógicos, los cuantificadores, paréntesis y comas, de las *variables* y de una *signatura* propia, la cual consiste de *constantes*, *funciones* y *predicados* o *relaciones*. En la tabla 5.4 presentamos una signatura para la lógica de *conjuntos* y en las tablas 5.5 y 5.6 presentamos dos signaturas para la *aritmética de Peano*, sin embargo, en la presentación ulterior de la aritmética de Peano supondremos la signatura de la tabla 5.5.

Si $\mathcal{L}$ es una lógica, se distinguen en ella palabras *bien formadas*:

Términos. Se forman recurrentemente por las siguientes reglas:

Constantes	:	$\emptyset$	-	conjunto vacío		
Funciones	:	$\bigcup$	-	unión (unaria)	:	$x \in \bigcup y \Leftrightarrow \exists z \in y : x \in y$
	:	$\mathcal{P}$	-	conjunto de partes (unaria)	:	$x \in \mathcal{P}(y) \Leftrightarrow \forall z(z \in x \Rightarrow z \in y)$
Relaciones	:	$\in$	-	pertenece a (binaria)		

Tabla 5.4: Signatura de la Teoría de Conjuntos.

Constantes	:	0	-	número cero
Funciones	:	s	-	sucesor (unaria)
Relaciones	:	$=$	-	igual a (binaria)

Tabla 5.5: Signatura AP de la Aritmética de Peano.

Constantes	:	Nullus	-	número cero
		I	-	número uno
		II	-	número dos
		$\vdots$		$\vdots$
		MCMXCVIII	-	número mil novecientos noventa y ocho
		$\vdots$		$\vdots$
		MMVIII	-	número dos mil ocho
		$\vdots$		$\vdots$
Funciones	:	*Más*	-	suma (binaria)
		Por	-	multiplicación (binaria)
Relaciones	:	$=$	-	igual a (binaria)
	:	$<$	-	menor que (binaria)

Tabla 5.6: Signatura AP_1 de la Aritmética de Peano.

1. $\xi \in \mathit{Variables} \cup \mathit{Constantes} \Rightarrow \xi \in \mathit{Términos}$.
2. $f \in \mathit{Funciones}^n, \xi_1, \ldots, \xi_n \in \mathit{Términos} \Rightarrow f(\xi_1, \ldots, \xi_n) \in \mathit{Términos}$.

Atomos. Símbolos de relaciones "evaluados" en términos:

1. $R \in \mathit{Relaciones}^n, \xi_1, \ldots, \xi_n \in \mathit{Términos} \Rightarrow R(\xi_1, \ldots, \xi_n) \in \mathit{Atomos}$.

Fórmulas. Se forman recurrentemente por las siguientes reglas:

1. $\phi \in \mathit{Atomos} \Rightarrow \phi \in \mathit{Fórmulas}$.
2. $\phi_1, \phi_2 \in \mathit{Fórmulas} \Rightarrow \neg\phi_1 \in \mathit{Fórmulas}$, $\phi_1 \vee \phi_2, \phi_1 \wedge \phi_2 \in \mathit{Fórmulas}$, $\phi_1 \to \phi_2, \phi_1 \leftrightarrow \phi_2 \in \mathit{Fórmulas}$
3. $\phi \in \mathit{Fórmulas} \Rightarrow \exists x\phi(x), \forall x\phi(x) \in \mathit{Fórmulas}$

Enunciados. Fórmulas sin variables "libres".

Demostrabilidad

En toda lógica hay *axiomas* los cuales son fórmulas de tipo *lógico* o de tipo *propio* de la teoría:

Axiomas lógicos. Cualesquiera que sean las fórmulas ϕ_1, ϕ_2, ϕ_3 y cualquiera que sea el término t, los siguientes son axiomas lógicos:

1. $\phi_1 \to (\phi_2 \to \phi_1)$.
2. $(\phi_1 \to (\phi_2 \to \phi_3)) \to ((\phi_1 \to \phi_2) \to (\phi_1 \to \phi_3))$.
3. $(\phi_1 \to \neg\phi_2) \to ((\phi_1 \to \phi_2) \to \neg\phi_1)$.
4. $\phi_1(t) \to \exists x\phi_1(x)$.

Sea *CP* (por *cálculo de predicados*) el conjunto de axiomas lógicos.

Axiomas propios. Son característicos de cada lógica. Por ejemplo, en la aritmética de Peano, se introduce la relación de desigualdad como una fórmula:

$$(x \leq y) \equiv \exists z(x + z = y).$$

Con esto, sea P^- el conjunto de los siguientes axiomas propios:

$s(x) \neq 0$	$x + 0 = x$
$x \neq 0 \Rightarrow s(0) \leq x$	$x + s(y) = s(x + y)$
$s(x) = s(y) \Rightarrow x = y$	$x + y = y + x$
	$(x + y) + z = x + (y + z)$
$x \cdot 0 = 0$	$(x \leq y) \vee (y \leq x)$
$x \cdot s(y) = x \cdot y + y$	$x \cdot (y + z) = x \cdot y + x \cdot z$
$x \cdot y = y \cdot x$	$x + y = x + z \Rightarrow y = z$
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	

Sea $AP = P^- \cup EI$, donde EI es el *esquema de inducción*:

Para cualquier fórmula Φ: $\Phi(0, \mathbf{x}) \wedge \forall x(\Phi(x, \mathbf{x}) \Rightarrow \Phi(s(x), \mathbf{x})) \Rightarrow \forall x(\Phi(x, \mathbf{x}))$.

En toda lógica se considera también *reglas de inferencia*. En el cálculo de predicados y en la aritmética de Peano, las reglas son de los tipos siguientes:

Modus Ponens. Cualesquiera que sean las fórmulas ϕ_1, ϕ_2: $$\frac{\phi_1 \to \phi_2 \quad \phi_1}{\phi_2}$$	**Generalización.** Cualquiera que sea la fórmula ϕ y cualquiera que sea la variable x pero siempre que x aparezca "libre" en ϕ: $$\frac{\phi(x)}{\forall x \phi(x)}$$

Sea H un conjunto de fórmulas bien formadas y sea ϕ una fórmula. Una *demostración* de ϕ a partir de H es una sucesión finita $D = [\delta_0, \ldots, \delta_m]$ de fórmulas bien formadas tal que

1. La última fórmula en D es la *tesis* ϕ, es decir, $\delta_m = \phi$.
2. Para cada fórmula δ en D se cumple una de las siguientes aseveraciones:
   (a) δ es un axioma, sea lógico o propio,
   (b) δ es una *hipótesis*, es decir, $\delta \in H$,
   (c) δ se sigue mediante reglas de inferencia de fórmulas que la preceden en D, es decir,
      i. $\exists \psi \in [\delta_0, \ldots, \delta[: (\psi \to \phi) \in [\delta_0, \ldots, \delta[$, o bien
      ii. $\exists \psi \in [\delta_0, \ldots, \delta[\, \exists x$ *variable libre en* $\psi : \phi = \forall x \psi(x)$.

La notación $H \vdash \phi$ denotará que existe una demostración de ϕ a partir de H, y en tal caso se dice que ϕ es *demostrable* a partir de H. Toda fórmula ϕ demostrable sin ningunas hipótesis suplementarias, $\emptyset \vdash \phi$, se dice ser un *teorema*, y se escribe en tal caso, $\vdash \phi$. De hecho seremos más enfáticos y escribiremos

- $CP \vdash \phi$ para indicar que ϕ se demuestra sólo de los axiomas lógicos, y en este caso se dice que ϕ es un teorema lógico,
- $P^- \vdash \phi$ para indicar que ϕ se demuestra de los axiomas lógicos y de los propios de la aritmética de Peano exceptuando el esquema de inducción, y
- $AP \vdash \phi$ para indicar que ϕ se demuestra en toda la aritmética de Peano.

Se tiene, por ejemplo, que todas las tautologías son teoremas lógicos así como también se puede demostrar que toda fórmula bien formada es equivalente a una fórmula escrita en *forma prenex*, es decir escrita como una fórmula donde todos los cuantificadores aparecen al principio y el *alcance* de los cuantificadores está en *forma normal conjuntiva*, o sea, es una conjunción de disyunciones de *literales*, las cuales son fórmulas atómicas o negaciones de fórmulas atómicas.

Dado un conjunto H de fórmulas bien formadas su *teoría* consta de todas las fórmulas demostrables a partir de H. La denotamos por *Teoría*$(H) = \{\phi | H \vdash \phi\}$. H se dice ser *consistente*, o más bien *coherente*, si no existe fórmula alguna ϕ tal que ambas ϕ y $\neg\phi$ están en *Teoría*(H).

Interpretaciones

Sea $\mathcal{L}$ una lógica. Una *estructura-*$\mathcal{L}$ es una pareja (E, Φ), donde E es un conjunto y Φ es una correspondencia que asocia

- a cada constante c de $\mathcal{L}$ un elemento $\Phi(c) \in E$ (es decir asocia "nombres" con "objetos",
- a cada símbolo de función f_m^n, de aridad n, le asocia una función $\Phi(f_m^n) : E^n \to E$, y
- a cada símbolo de relación R_m^n, de aridad n, le asocia una relación $\Phi(R_m^n) \subset E^n$.

Por ejemplo, $(\mathbb{N}, id)$ es una estructura-AP, donde $\mathbb{N}$ es el conjunto de números naturales e id es la interpretación usual de los símbolos aritméticos. También, el conjunto de funciones de los naturales en los naturales $\mathbf{N} = \mathbb{N}^{\mathbb{N}}$ es una estructura-AP con la interpretación de los símbolos "punto-a-punto":

Cero 0	:	$\Phi(0) = \mathbf{0}$, donde $\mathbf{0} : n \mapsto 0$ es la función cero,
Sucesor s	:	$\Phi(s) : f \mapsto f'$, donde $f' : n \mapsto s(f(n))$ es la función $f + 1$,
Suma $+$	:	$\Phi(+) : (f, g) \mapsto f + g$, donde $f + g : n \mapsto f(n) + g(n)$ es la función suma punto-a-punto,
Igualdad $=$	:	$\Phi(=) = \{(f, g) \mid \forall n : f(n) = g(n)\}$, es decir, dos funciones son iguales si lo son sus correspondientes valores en cada punto.

Mencionemos una tercer estructura de AP. Para esto es menester introducir algunos tecnicismos. Un *filtro* en $\mathbb{N}$ es una colección de conjuntos "grandes", es decir, es un conjunto $\mathcal{F} \subset \mathcal{P}(\mathbb{N})$ tal que

$$\begin{aligned} &\text{i)} && \mathbb{N} \in \mathcal{F} \\ &\text{ii)} && A \in \mathcal{F}, A \subset B \;\Rightarrow\; B \in \mathcal{F} \\ &\text{iii)} && A, B \in \mathcal{F} \;\Rightarrow\; A \cup B \in \mathcal{F} \end{aligned}$$

por ejemplo, el *filtro de Frèchet* $\mathcal{F} = \{\{m \in \mathbb{N} | m \geq n\}\}_{n \in \mathbb{N}}$ es, en efecto, un filtro. Un *ultrafiltro* es un filtro $\mathcal{U}$ consistente de los conjuntos "más grandes posibles", es decir, además de satisfacer los axiomas de filtro, satisface también

$$\text{iv)} \quad A \in \mathcal{P}(\mathbb{N}) \;\Rightarrow\; (A \in \mathcal{U}) \vee (A^c \in \mathcal{U})$$

Bajo ciertas suposiciones[1] es posible ver que todo filtro se extiende a un ultrafiltro (de hecho, a una infinidad de ultrafiltros). Sea pues $\mathcal{U}$ un ultrafiltro en $\mathbb{N}$ y sea $\sim_{\mathcal{U}}$ la relación en $\mathbf{N}$ definida como

$$f \sim_{\mathcal{U}} g \;\Leftrightarrow\; \{n \in \mathbb{N} | f(n) = g(n)\} \in U.$$

$\sim_{\mathcal{U}}$ es una relación de equivalencia congruente con las operaciones de $\mathbf{N}$ y por tanto $\mathbb{N}^* = \mathbf{N}/\sim_{\mathcal{U}}$ es una estructura-AP.

Si (E, Φ) es una estructura-$\mathcal{L}$ las fórmulas atómicas cerradas, es decir, aquellas que no tienen variables libres, de $\mathcal{L}$ se evalúan naturalmente en E. A partir de ellas se puede evaluar a todas los *enunciados*, es decir, fórmulas bien formadas que no tienen variables libres:

- $\phi = \neg\phi_1$ es *verdadera* en E si ϕ_1 es falsa en E,
- $\phi = (\phi_1 \to \phi_2)$ es *verdadera* en E si dado que ϕ_1 es verdadera en E necesariamente ha de tenerse que ϕ_2 es verdadera en E,
- $\phi = \forall x \phi_1(x)$ es *verdadera* en E si para cualquier sustitución de x por un elemento e en E, $\phi_1(e)$ es verdadera en E.

Si un enunciado ϕ es verdadero en E escribiremos $E \models \phi$, y en tal caso diremos que E es una *interpretación* o *modelo* de ϕ. Si H es un conjunto de enunciados, escribiremos $E \models H$ si para cada $\phi \in H$ se cumple $E \models \phi$.

Por ejemplo, la fórmula de la aritmética de Peano

$$\phi_0 \equiv \forall x (x \neq 0 \to \exists y (s(y) = x)$$

que asevera que todo elemento no nulo posee un antecesor es válida en $\mathbb{N}$, no lo es en $\mathbf{N}$ pues la función $x \mapsto (x \bmod 2)$ es no-nula mas no posee antecesor alguno. Sin embargo, ϕ_0 sí es verdadera en $\mathbb{N}^*$.

ϕ es *universalmente válido* si para cualquier estructura-$\mathcal{L}$ (E, Φ) se cumple $E \models \phi$.

[1] A saber, bajo la aceptación del *Principio de Selección* o, equivalentemente, del *Lema de Zorn*. Por tanto, esta suposición no puede ser aceptada de manera constructiva.

Si H es un conjunto de enunciados y ϕ es otro enunciado, diremos que ϕ es una *consecuencia lógica* de ϕ, y escribiremos $H \models \phi$, si rige la implicación siguiente:

$$\forall E : (E \text{ estructura-}\mathcal{L}) \;\Rightarrow\; (E \models H \;\Rightarrow\; E \models \phi).$$

La lógica $\mathcal{L}$ se dice ser

sólida si para cualquier conjunto de enunciados H y cualquier otro enunciado ϕ rige la implicación

$$H \vdash \phi \;\Rightarrow\; H \models \phi,$$

completa si para cualquier conjunto de enunciados H y cualquier otro enunciado ϕ rige la implicación recíproca

$$H \models \phi \;\Rightarrow\; H \vdash \phi.$$

Ambos el cálculo de predicados y la aritmética de Peano son sólidos. El *Teorema de Completitud de Gödel* asevera que cálculo de predicados es también completo. Su demostración excede los alcances del presente texto y por eso omitimos su demostración.

Incompletitud

Veremos aquí que la aritmética de Peano no es completa. Para esto, codificaremos a la aritmética dentro de la misma aritmética y propondremos un enunciado tal que ni él ni su negación son demostrables en AP.

Para cada $n \in \mathbb{N}$ construímos el *n-ésimo numeral* como $\mathbf{n} = s^n(0) = \underbrace{s(\cdots s(}_{n \text{ veces}} 0)\cdots)$. Una relación $A \subset \mathbb{N}^k$ se dice *representable* en AP si existe una fórmula bien formada $\phi_A(x_1, \ldots, x_k)$ tal que para cualesquiera $n_1, \ldots, n_k \in \mathbb{N}$ se tiene que rigen las implicaciones

$$\begin{aligned} (n_1, \ldots, n_k) \in A &\;\Rightarrow\; AP \vdash \phi_A(\mathbf{n}_1, \ldots, \mathbf{n}_k) \\ (n_1, \ldots, n_k) \notin A &\;\Rightarrow\; AP \vdash \neg\phi_A(\mathbf{n}_1, \ldots, \mathbf{n}_k) \end{aligned}$$

Una función $\mathbb{N}^n \to \mathbb{N}$ es *representable* si su gráfica[2] lo es.

El concepto de *demostrabilidad* es muy algorítmico, por lo cual una demostración puede verse como un método de cálculo o de comprobación. Así pues, el resultado siguiente aparece de manera muy natural, aunque en este texto omitiremos su demostración.

Teorema 5.4.1 (Gödel) *Una función es representable si y sólo si es recursiva. Una relación es representable si y sólo si es recursiva.*

Ahora, es evidente que el conjunto de fórmulas bien formadas es efectivamente numerable, y cualquiera de los métodos ya vistos de enumeración basta para corroborar esta afirmación. De manera alternativa, sin embargo, podríamos introducir la enumeración g que se muestra en la tabla 5.7. Ahí observamos que los símbolos del alfabeto quedan codificados por números impares, todas las cadenas por números pares divisibles por una potencia impar de 2 y todas las cadenas de cadenas por números pares divisibles por una potencia par de 2.

Definamos las siguientes relaciones en $\mathbb{N}$:

1. *Fbf* (Fórmula bien formada): $Fbf(n) \;\Leftrightarrow\; \exists\phi$ fórmula bien formada : $n = g(\phi)$
2. *AxL* (Axioma lógico): $AxL(n) \;\Leftrightarrow\; \exists\phi$ fórmula bien formada : (ϕ es axioma lógico) $\wedge$ ($n = g(\phi)$)

[2] Es decir, ella misma.

Especiales:	(	$\mapsto$	3
	)	$\mapsto$	5
	,	$\mapsto$	7
	$\neg$	$\mapsto$	9
	$\rightarrow$	$\mapsto$	11
	$\forall$	$\mapsto$	13
Variables:	x_j	$\mapsto$	$7 + 8j$
Constantes:	c_j	$\mapsto$	$9 + 8j$
Funciones:	f_j^n	$\mapsto$	$11 + 8(2^n \cdot 3^j)$
Relaciones:	R_j^n	$\mapsto$	$13 + 8(2^n \cdot 3^j)$
Cadenas:	$\xi_0\xi_1\cdots\xi_k$	$\mapsto$	$2^{g(\xi_1)}3^{g(\xi_2)}\cdots p_k^{g(\xi_k)}$
Cadenas de cadenas:	$\sigma_0\sigma_1\cdots\sigma_k$	$\mapsto$	$2^{g(\sigma_1)}3^{g(\sigma_2)}\cdots p_k^{g(\sigma_k)}$

(donde p_k es el k-ésimo número primo.)

Tabla 5.7: Enumeración de Gödel.

3. *AxP* (Axioma propio): $AxP(n) \Leftrightarrow \exists\phi$ fórmula bien formada : (ϕ es axioma propio) $\wedge$ ($n = g(\phi)$)

4. *Demos* (Demostración):

$$Demos(n) \Leftrightarrow \exists\Delta \in (\text{fórmula bien formada})^* : (\Delta \text{ es una demostración}) \wedge (n = g(\Delta))$$

5. *Dmble* (Demostrable):

$$Dmble(n) \Leftrightarrow \exists\Delta\exists m : (m = g(\Delta)) \wedge Demos(m) \wedge (n = (\text{exponente del máximo primo que divide a } m))$$

6. *Sust* (Sustitución): $Sust(m, n, p, q) \Leftrightarrow \exists\phi, x, t :$ ϕ es una fórmula que involucra a la variable x, t es un término, $n = g(\phi), p = g(x), q = g(t)$, y $m = g(\phi(x|t))$, donde $\phi(x|t)$ es la fórmula que se obtiene al sustituir toda aparición de x por el término t.

7. *Inst* (Instanciación): $Inst(m, n) \Leftrightarrow \exists\phi(x) : n = g(\phi(\mathbf{m}))$.

8. *Cump* (Cúmplese): $Cump(m, n) \Leftrightarrow \exists\Delta, \phi(x) : \Delta$ es una prueba de $\phi(\mathbf{m})$, y $n = g(\Delta)$.

Proposición 5.4.1 *Las relaciones anteriores son todas recursivas y por ende son representables en AP.*

Utilizaremos los mismos nombres para denotar a los predicados que representan a esas relaciones. Tenemos entonces que para cualesquiera $m, n \in \mathbb{N}$ se cumplen las equivalencias siguientes:

$AP \vdash Cump(\mathbf{m}, \mathbf{n})$	$\Leftrightarrow$	$Cump(m, n)$ se cumple	$\Leftrightarrow$	para alguna fórmula $\phi(x)$, n codifica una demostración de $\phi(\mathbf{m})$,
$AP \vdash \neg Cump(\mathbf{m}, \mathbf{n})$	$\Leftrightarrow$	$Cump(m, n)$ no se cumple	$\Leftrightarrow$	n no codifica demostración alguna de $\phi(\mathbf{m})$, para ninguna fórmula $\phi(x)$.

Hagamos

$$\alpha_1(x_1) \equiv \forall x_2(\neg Cump(x_1, x_2)) \ : \ \text{ningún “enunciado” de la forma } \phi(x_1) \text{ es demostrable,}$$

y sean $p = g(\alpha_1(x_1))$ y $\alpha \equiv \alpha_1(\mathbf{p})$.

Se ve pues que α asegura que ningún enunciado de la forma $\phi(\mathbf{p})$ es demostrable, en particular, para $\phi = \alpha_1$, asevera que $\alpha_1(\mathbf{p})$ no es demostrable, es decir, que α mismo no es demostrable. En otras palabras α es un enunciado que se refiere a sí mismo y asevera su propia indemostrabilidad.

De aquí resulta que si AP fuese consistente, entonces $AP \nvdash \alpha$.

Una teoría T, que posea un modelo que incluya a $\mathbb{N}$, es *consistente-ω* si para cualquier fórmula $\phi(x)$, dado que para cada $n \in \mathbb{N}$ se tuviera que $T \vdash \phi(\mathbf{n})$ entonces necesariamente $T \nvdash \neg\forall x : \phi(x)$.

Se observa que si T es consistente-ω entonces es también consistente, a secas.

Si AP fuese consistente-ω, entonces $AP \nvdash \neg\alpha$.

Con todo esto se tiene el

Teorema 5.4.2 (Primero de Incompletitud de Gödel) *Si AP fuese consistente-ω entonces ha de ser incompleta.*

Vemos muchas semejanzas entre este teorema de incompletitud y el problema de la parada de máquinas de Turing. En última instancia, las imposibilidades aseveradas por ellos se demuestran utilizando mecanismos autoreferentes. Por otro lado, la capacidad de construir esos mecanismos se debe a sus propios niveles de expresividad y a sus características procedimentales.

Concluímos esta sección presentando el

Teorema 5.4.3 (Segundo de Incompletitud de Gödel) *En AP no puede demostrarse su propia consistencia.*

De manera muy resumida, tenemos los hechos siguientes:

1. Si H es un conjunto inconsistente de fórmulas bien formadas entonces la teoría de H, *Teoría*(H), coincide con el conjunto de todas las fórmulas bien formadas. En otras palabras, en una teoría inconsistente cualquier cosa es demostrable.

2. Un enunciado ϕ, con número de Gödel $n = g(\phi)$, es demostrable en AP si y sólo si $AP \vdash \mathit{Dmble}(\mathbf{n})$ y es indemostrable en AP si y sólo si $AP \vdash \neg\mathit{Dmble}(\mathbf{n})$.

3. En AP se puede mostrar que 0 es distinto a $1 = s(0)$. La negación de esta desigualdad es $\phi_0 \equiv 0 = s(0)$. Así pues, tendremos que AP es consistente si y sólo si no se puede demostrar ϕ_0. Sea pues $n_0 = g(\phi_0)$ y sea $\mathit{Con} \equiv \neg\mathit{Dmble}(\mathbf{n}_0)$. El Segundo Teorema de Incompletitud de Gödel equivale a mostrar queen efecto $AP \nvdash \neg\mathit{Dmble}(\mathbf{n}_C)$, donde $n_C = g(\mathit{Con})$.

4. De acuerdo con la regla *modus ponens*, si $n_{12} = g(\phi_1 \to \phi_2)$, $n_1 = g(\phi_1)$ y $n_2 = g(\phi_2)$ entonces

$$AP \vdash (\mathit{Dmble}(\mathbf{n}_{12}) \wedge \mathit{Dmble}(\mathbf{n}_1) \to \mathit{Dmble}(\mathbf{n}_2))$$

5. Además tenemos que todo lo demostrable es demostrablemente demostrable, es decir, si $n_1 = g(\phi_1)$ y $n_2 = g(\mathit{Dmble}(\mathbf{n}_1))$ entonces

$$AP \vdash (\mathit{Dmble}(\mathbf{n}_1) \to \mathit{Dmble}(\mathbf{n}_2))$$

6. Por todo lo anterior, para probar el Segundo Teorema de Incompletitud basta ver que *Con* es demostrablemente equivalente en AP al enunciado α demostrado indemostrable en el Primer Teorema de Incompletitud.

   Utilicemos la notación $\phi(\ulcorner\xi\urcorner)$ para denotar a la fórmula $\phi(\mathbf{n})$ donde $n = g(\xi)$. Esquemáticamente tenemos:

Veamos primero $AP \vdash \alpha \to Con$:

$$\begin{aligned} AP \vdash (0 = s(0)) \to \alpha &\Rightarrow AP \vdash Dmble(\ulcorner (0 = s(0)) \to \alpha \urcorner) \\ &\Rightarrow AP \vdash Dmble(\ulcorner (0 = s(0)) \urcorner) \to Dmble(\ulcorner \alpha \urcorner) \\ &\Rightarrow AP \vdash \neg Dmble(\ulcorner \alpha \urcorner) \to \neg Dmble(\ulcorner (0 = s(0)) \urcorner) \\ &\Rightarrow AP \vdash \alpha \to Con \end{aligned}$$

pues $AP \vdash \alpha \to \neg Dmble(\ulcorner \alpha \urcorner)$.

Recíprocamente, veamos $AP \vdash Con \to \alpha$:

$$\begin{aligned} AP \vdash Dmble(\ulcorner \alpha \urcorner) \to Dmble(\ulcorner Dmble(\ulcorner \alpha \urcorner) \urcorner) &\Rightarrow AP \vdash Dmble(\ulcorner \alpha \urcorner) \to Dmble(\ulcorner \neg\alpha \urcorner) \\ &\Rightarrow AP \vdash Dmble(\ulcorner \alpha \urcorner) \to Dmble(\ulcorner \alpha \wedge \neg\alpha \urcorner) \\ &\Rightarrow AP \vdash Dmble(\ulcorner \alpha \urcorner) \to Dmble(\ulcorner 0 = s(0) \urcorner) \\ &\Rightarrow AP \vdash \neg Dmble(\ulcorner 0 = s(0) \urcorner) \to \neg Dmble(\ulcorner \alpha \urcorner) \\ &\Rightarrow AP \vdash Con \to \alpha \end{aligned}$$

5.4.2 El teorema de Goodstein

Los teoremas que hemos visto hasta ahora indemostrables en la aritmética de Peano son enunciados autoreferentes. Veremos en esta sección algunos teoremas de "tipo aritmético" que son independientes en la aritmética de Peano, es decir, son tales que ni ellos ni sus respectivas negaciones son demostrables en AP.

Representación formal en base m

Sea $m \geq 2$. Para cada $a \in N$ definimos $rp_m(a)$ como sigue:

$$\begin{aligned} \text{i)}\quad a \leq m^m - 1 &\Rightarrow \exists! a_0, \ldots, a_{m-1} \in [0, m-1] : \\ &\qquad a = \textstyle\sum_{i=0}^{m-1} a_i \cdot m^i \\ &\Rightarrow rp_m(a) = \textstyle\sum_{i=0}^{m-1} a_i \cdot m^i \\ \text{ii)}\quad a \geq m^m &\Rightarrow \exists! a_0, \ldots, a_{\lceil \log_m(a) \rceil - 1} \in [0, m-1] : \\ &\qquad a = \textstyle\sum_{i=0}^{\lceil \log_m(a) \rceil - 1} a_i \cdot m^i \\ &\Rightarrow rp_m(a) = \textstyle\sum_{i=0}^{\lceil \log_m(a) \rceil - 1} a_i \cdot m^{rp_m(i)} \end{aligned}$$

Ejemplo. Para $m = 2$ y $a = 100$ tenemos

$$a = 1100100_2 = 2^6 + 2^5 + 2^2,$$

luego

$$rp_2(100) = 2^{2^2+2} + 2^{2^2+1} + 2^2.$$

Escribamos $rp(a, m) = rp_m(a)$. Con esta notación denotaremos por $rp(a, k|m)$ al resultado de sustituir a m por k en $rp(a, m)$.

Sucesión de Goodstein

Para $a \in N$ y $m \geq 2$ dados, definiremos a la *sucesión de Goodstein* $S(a, m) = \{s_n(a, m)\}_{n \geq 0}$ de manera recursiva. Para ello nos auxiliaremos de la sucesión $B(a, m) = \{b_n(a, m)\}_{n \geq 0}$ definida igualmente de manera

recursiva. Explícitamente, hacemos

$$\begin{aligned} & b_0 = m && ; \quad s_0 = a \\ \forall n \geq 0: \quad & b_{n+1} = b_n + 1 && ; \quad s_{n+1} = rp(s_n, b_{n+1}|b_n) - 1 \end{aligned}$$

Ejemplo. Para $m = 2$ y $a = 100$ tenemos $rp_2(100) = 2^{2^2+2} + 2^{2^2+1} + 2^2$. Luego

$$\begin{aligned} s_0(100,2) &= 2^{2^2+2} + 2^{2^2+1} + 2^2 \\ &= 100 \\ s_1(100,2) &= 3^{3^3+3} + 3^{3^3+1} + 3^3 - 1 \\ &= 3^{3^3+3} + 3^{3^3+1} + 2 \cdot 3^2 + 2 \cdot 3 + 2 \\ &= O(3^{27}) \\ s_2(100,2) &= 4^{4^4+4} + 4^{4^4+1} + 2 \cdot 4^2 + 2 \cdot 4 + 1 \\ &= O(2^{512}) \\ \vdots & \quad \vdots \end{aligned}$$

Sorpresivamente tenemos que ... ¡la sucesión de Goodstein converge a cero!

Teorema de Goodstein: $\forall a \forall m \exists n: \ s_n(a,m) = 0$.

- el teorema se cumple en $\mathbb{N}$: el modelo estándar de los números naturales, sin embargo
- no es demostrable en la aritmética de Peano,
  - en términos de (a,m), el mínimo n que anula a la sucesión de Goodstein crece vertiginosamente rápido.

Función de Goodstein: Definimos a la función G como

$$G : (a,m) \mapsto G : (a,m) = \text{Min}\{n | s_n(a,m) = 0\}.$$

Puede probarse que la *diagonal de* G "domina" a cualquier función recursiva, es decir:

$$\forall f \in \{\text{funciones recursivas}\} \exists n_0: \ n \geq n_0 \ \Rightarrow \ f(n) < G(n,n).$$

Se tiene:

- G no puede ser recursiva y
- el Teorema de Goodstein no es demostrable formalmente, pues
  - una demostración formal de él daría un algoritmo para calcular G, y no puede existir tal algoritmo pues la función que calcularía tal algoritmo estaría acotada a la larga por G.

Ilustración del Teorema de Goodstein

Supongamos que a es de la forma $a = a_i \cdot m^i$, donde $i < m$, con $a_i < m$. Al calcular la sucesión de Goodstein, el término que le sigue a a es

$$\begin{aligned} a_i \cdot (m+1)^i - 1 &= (a_i - 1) \cdot (m+1)^i + \\ &\quad (m+1)^i - 1 \\ &= (a_i - 1) \cdot (m+1)^i + \\ &\quad m \cdot (m+1)^{i-1} + \cdots + m \cdot (m+1) + m \end{aligned}$$

así pues

- se ha decrementado en uno el dígito más significativo de la expresión formal, aunque
- se han "restablecido" los demás dígitos a sus máximos valores posibles. Sin embargo,
  - estos dígitos son estrictamente menores que la base actual,
  - no se modificarán más al calcular la sucesión: ¡Sólo pueden decrecer!
- Al cabo de un cierto número de iteraciones los dígitos menos significativos se anularán.
- En este momento el término actual es de la forma inicial pero se ha decrementado en una unidad el dígito más significativo.

Algunos valores explícitos de la función

Calcularemos valores para argumentos de la forma $(a \cdot m^i, m)$ donde $a \in [1, m-1]$, $i \in [0, m-1]$ y $m \geq 2$.

Definamos

$$\begin{aligned} p(i,a,m) &= \text{Min}\{k | s_k(a \cdot m^i, m) = 0\}, \\ q(i,a,m) &= \text{Min}\{k | s_k(a \cdot m^i, m) = (a-1) \cdot (m+k)^i\}. \end{aligned}$$

Entonces $q(0,a,m) = 1 \qquad p(0,a,m) = a$

Recursivamente, para $i \geq 1$, los valores de q y de p se calculan como sigue

$s := 1$; **for** $j = 0$ **to** $i-1$ **do** $\{r := p(j,m,m+s)$; $s := s + r$ $\}$; $q(i,a,m) := s$	$s := 0$; **for** $b = 0$ **to** $a-1$ **do** $\{r := q(i,a-b,m+s)$; $s := s + r$ $\}$; $p(i,a,m) := s$

Equivalentemente, se tiene las recurrencias siguientes:

$$\left\{ \begin{array}{rcl} q(0,a,m) &=& 1 \\ q(i,a,m) &=& q(i-1,a,m) + p(i-1,m,m+q(i-1,a,m)) \end{array} \right.$$

y

$$\left\{ \begin{array}{rcl} p(0,a,m) &=& a \\ \left\{ \begin{array}{rcl} p(i,1,m) &=& q(i,1,m) \\ p(i,a,m) &=& q(i,a,m) + p(i,a-1,m+q(i,a,m)) \end{array} \right. & & \end{array} \right.$$

De aquí se puede ver que q es "constante respecto a m", $\forall a : \; q(i,a,m) = q(i,1,m)$.

Resolución del sistema de recurrencias

Proposición 5.4.2 *Supongamos que $h_i : N \to N$ es una función tal que $\forall a, m :\ q(i, a, m) = h_i(m) - m$.*

Entonces

1. $p(i, a, m) = h_i^a(m) - m$.
2. $q(i + 1, a, m) = h_i^{m+1}(m) - m$.

Demostración: Probemos 1) por inducción en a:

Caso $a = 1$:

$$\begin{aligned} p(i, 1, m) &= q(i, 1, m) \\ &= h_i^1(m) - m \end{aligned}$$

Caso $a + 1$:

Sea $a \geq 1$. Supongamos que $p(i, a, m) = h_i^a(m) - m$. De acuerdo a la recurrencia localizada para p tenemos

$$\begin{aligned} p(i, a + 1, m) &= q(i, a + 1, m) + p(i, a, m + q(i, a + 1, m)) \\ &= (h_i(m) - m) + (p(i, a, h_i(m))) \\ &= (h_i(m) - m) + (h_i^a(h_i(m)) - h_i(m)) \\ &= h_i^{a+1}(m) - m \end{aligned}$$

Probemos ahora 2).

De la recurrencia de q obtenemos

$$\begin{aligned} q(i + 1, a, m) &= q(i, a, m) + p(i, m, m + q(i, a, m)) \\ &= (h_i(m) - m) + (p(i, m, h_i(m))) \\ &= (h_i(m) - m) + (h_i^m(h_i(m)) - h_i(m)) \\ &= h_i^{m+1}(m) - m \end{aligned}$$

Así pues podemos definir

$$\begin{aligned} h_{i+1} : N &\to N \\ m &\mapsto h_{i+1}(m) = h_i^{m+1}(m) \end{aligned}$$

Ya que $q(0, a, m) = 1$ tenemos $h_0(m) = m + 1$.

En resumen, al definir a la sucesión $(h_i)_{i\geq 0}$ haciendo

$$\begin{aligned} h_0(m) &= m + 1 \\ \forall i \geq 1 :\ h_{i+1}(m) &= h_i^{m+1}(m) \end{aligned}$$

las funciones q y p quedan de la forma

$$\begin{aligned} q(i, a, m) &= h_i(m) - m \\ p(i, a, m) &= h_i^a(m) - m \end{aligned}$$

Ejemplos. Tan sólo para ilustrar los crecimientos de las funciones h_i observamos:

$$\begin{aligned} h_0(m) &= m+1 \\ h_1(m) &= 2m+1 \\ h_2(m) &= 2^{m+1}(m+1)-1 \end{aligned}$$

Para ilustrar el crecimiento de h_3 vemos las primeras iteraciones de h_2. Se tiene

$$\begin{aligned} h_2^2(m) &= 2^{1+m+2^{1+m}(1+m)}\,(1+m)-1 \\ h_2^3(m) &= 2^{1+m+2^{1+m}(1+m)+2^{1+m+2^{1+m}(1+m)}(1+m)}\,(1+m)-1 \\ h_2^4(m) &= 2^{1+m+2^{1+m}(1+m)+2^{1+m+2^{1+m}(1+m)}(1+m)+2^{1+m+2^{1+m}(1+m)+2^{1+m+2^{1+m}(1+m)}(1+m)}(1+m)}\,(1+m)-1 \end{aligned}$$

Pues bien

$$h_3(m) = h_2^{m+1}(m).$$

Los crecimientos de h_i con $i \geq 4$ son ciertamente inimaginables.

$G(a,m)$ **con** $a < m^m$

Hemos visto que si a es de la forma $a = a_i \cdot m^i$ entonces $G(a,m) = p(i,a_i,m)$.

Ahora, si $a < m^m$ tenemos que $\exists a_0,\ldots,a_{m-1} \in [0,m-1]$ tales que $a = \sum_{i=0}^{m-1} a_i \cdot m_i$.

En este caso, $G(a,m)$ es el número de veces que hay que aplicar la construcción de Goodstein para anular a todos los dígitos a^i. Teniendo en cuenta que en cada iteración se incrementa la base, vemos que $G(a,m)$ se calcula mediante el algoritmo siguiente:

$s := 0$;
for $i = 0$ **to** $m-1$ **do**
    $\{r := p(i, a_m, m+s)$;
    $s := s + r$ $\}$;
$G(a,m) := s$

5.4.3 La conjetura de Catalan

Motivación

El Teorema de Goodstein proporciona un ejemplo de una función con un muy rápido crecimiento que responde afirmativamente una pregunta. Con ella, encontramos números muy grandes que "testifican" la veracidad de algunas proposiciones.

Veremos aquí que algunos números muy grandes son límites para buscar candidatos que "testifiquen" una proposición. Sin embargo son tan grandes que prácticamente coinciden con "infinito".

Formulación

En la primera mitad del siglo XIX Catalan conjeturó que las únicas potencias sucesivas en los naturales son 8 y 9. En otras palabras:

$$\forall x,n,y,m:\ x^n - y^m = 1 \Leftrightarrow x = 3 \wedge n = 2 \wedge y = 2 \wedge m = 3.$$

Sea $N_2 = \{x \in \mathbb{N} | n \geq 2\}$.

Hasta ahora la conjetura de Catalan ha permanecido abierta.

Sin embargo ha habido "algunos avances".

Proposición 5.4.3 (Tijdeman, 1970) *Si* $(x, n, y, m) \in N_2^4$ *es una solución de la ecuación de Catalan entonces existe una constante* C *efectivamente computable, tal que* $\mathrm{Max}(x, n, y, m) \leq C$.

El problema entonces es estimar la constante C.

Proposición 5.4.4 (Langevin, 1976) *La constante* C *satisface la desigualdad* $C \leq \exp(\exp(\exp(\exp(730))))$.

Ya que $\exp(x) = 10^{\log(e^x)} = 10^{x \cdot \log(e)}$ y $\log(e) = 0.4342944819033\ldots$ se tiene que

$\exp(730)$	se escribe (en decimal) con casi $\frac{730}{2}$ dígitos,
$\exp(\exp(730))$	tiene una longitud (en decimal) que se escribe con casi $\frac{730}{2}$ dígitos,
$\exp(\exp(\exp(730)))$	tiene una longitud (en decimal) cuya longitud se escribe con casi $\frac{730}{2}$ dígitos,
$\exp(\exp(\exp(\exp(730))))$	tiene una longitud (en decimal) cuya longitud tiene una longitud cuya longitud se escribe con casi $\frac{730}{2}$ dígitos,

El número de átomos en el Universo es "apenas" del orden de 10^{100}. Luego, si contamos un "Universo" por cada átomo del Universo tendremos 10^{200} átomos. A esta cantidad habría que multiplicarla por 10^{65} para obtener sólo la primera cantidad de la lista anterior.

Proposición 5.4.5 (Baker, 1982) *Para soluciones* x, n, y, m *de la ecuación de Catalan, con los cuatro números mayores o iguales que 3, se ha de tener necesariamente que*

$$\mathrm{Max}(x, y) \leq \mathrm{Min}\left\{\exp\left(\exp\left((5n)^{10} m^{10m^3}\right)\right), \exp\left(\exp\left((5m)^{10} n^{10n^3}\right)\right)\right\}.$$

Fijos n y m, para buscar las posibles soluciones (x, y) utilizando una CRAY se necesitaría una cantidad de tiempo comparada con la cual toda la edad del Universo equivaldría a un mero parpadeo.

Sin embargo, ¡¡todo lo involucrado es computable!!

5.5 Clasificación de problemas irresolubles

5.5.1 Computaciones con oráculos

Sea $G : \mathbb{N} \to \mathbb{N}$ una función. Los *programas-G-***while** se construyen como los programas-**while** considerando como primitiva a la instrucción

$$\langle \mathit{Variable} \rangle := G(\langle \mathit{Variable} \rangle),$$

la que se dice ser un *oráculo* para la función G.

Una función $f : \mathbb{N} \to \mathbb{N}$ es *computable-G* si es calculable por un programa-G-**while**. La clase de funciones recursivas *relativas al oráculo* G se define como

$$\mathit{Rec}(G) = \{f : \mathbb{N} \to \mathbb{N} | f \text{ es computable-}G\}.$$

Proposición 5.5.1 *Las siguientes propiedades se cumplen (evidentemente):*

1. *Si f es recursiva entonces $f \in Rec(Id)$, donde Id es la función identidad.*
2. *Si f es recursiva y G es cualquier función total entonces $f \in Rec(G)$. En otras palabras, las funciones computables lo son también relativas a cualquier otra función total.*
3. *Si G es total entonces $G \in Rec(G)$.*
4. *Si $F \in Rec(G)$ y $G \in Rec(H)$ entonces $F \in Rec(H)$.*
5. *$G \in Rec \Leftrightarrow Rec = Rec(G)$.*

Proposición 5.5.2 *Las clases siguientes son efectivamente numerables:*

1. *La clase de programas-G-***while**.
2. *La clase $Rec(G)$ de funciones computables-G.*

De aquí resulta el

Teorema 5.5.1 (de Universalidad Relativizada) *Sea $\{f_{G,n}\}_{n\geq 0}$ una enumeración de las funciones computables-G. Si G es total entonces la función $U_G : (n, m) \mapsto f_{G,n}(m)$ es universal para $Rec(G)$, y está en la misma clase $Rec(G)$.*

Naturalmente, se tiene el

Problema de la Palabra Relativizado: *Sea $\{f_{G,n}\}_{n\geq 0}$ una enumeración de las funciones computables-G. Definamos*

$$\begin{array}{rcl} P_G : \mathbb{N}^2 & \to & N \\ (n, m) & \mapsto & \begin{cases} 1 & \text{si } f_{G,n}(m)\downarrow, \\ 0 & \text{en otros casos.} \end{cases} \end{array}$$

Entonces $P_G \notin Rec(G)$.

5.5.2 Enumerabilidad recursiva relativa a oráculos

En esta parte introduciremos algunas nociones básicas que presentaremos (Cfr 7.1.1) más adelante. Veremos aquí que los problemas irresolubles mediante funciones computables son heredados por clases de funciones que extienden a las computables mediante *oráculos*.

Dados dos conjuntos A, B, $B \neq \emptyset$, diremos que

- A es *recursivo-B* si $\chi_A \in Rec(\chi_B)$. En este caso, escribiremos $A \preceq_r B$.
- A es *reducible múltiplemente a B* si existe una función computable f tal que $A = f^{-1}(B)$, es decir, si

  $$\forall x : \ x \in A \ \Leftrightarrow \ f(x) \in B.$$

  En este caso, escribiremos $A \preceq_m B$.
- A es *reducible mónicamente a B* si existe una función computable e inyectiva f tal que $A = f^{-1}(B)$, en este caso, escribiremos $A \preceq_1 B$.

De manera natural se tiene que las siguientes proposiciones son válidas:

Proposición 5.5.3 *La relación "$\preceq_r$" es reflexiva y transitiva.*

En efecto, es reflexiva porque la función identidad es computable. Es transitiva, porque la clase de funciones computables es cerrada bajo composición de funciones.

Proposición 5.5.4 *"$\preceq_1$" es un refinamiento de "$\preceq_m$" y "$\preceq_m$" es un refinamiento de "$\preceq_r$", es decir*

$$A \preceq_1 B \;\Rightarrow\; A \preceq_m B \qquad ; \qquad A \preceq_m B \;\Rightarrow\; A \preceq_r B$$

En efecto, la condición que define a "$\preceq_1$" es más fuerte que la de "$\preceq_m$", pues aquella exige que la función f tal que $A = f^{-1}(B)$ sea inyectiva. por otro lado, si $A = f^{-1}(B)$, donde f es computable, entonces $\chi_A = \chi_B \circ f$, es decir χ_A es recursiva respecto a χ_B.

Proposición 5.5.5 *Las relaciones "$\preceq_1$" y "$\preceq_m$" son reflexivas y transitivas.*

En efecto, la función identidad es computable y la composición de computables es computable.

Toda relación entre conjuntos que sea reflexiva y transitiva se dice ser una *reducibilidad*.

Sea $\mathcal{C}$ una clase de conjuntos y "$\preceq$" una reducibilidad.

- La clase $\mathcal{C}$ es *cerrada* respecto a la reducibilidad "$\preceq$" si $\forall A, B:\ (B \in \mathcal{C})\&(A \preceq B) \Rightarrow (A \in \mathcal{C})$.
- Un conjunto B es *difícil* para la clase $\mathcal{C}$ si $\forall A:\ (A \in \mathcal{C}) \Rightarrow (A \preceq B)$.
- Un conjunto B es *completo* para la clase $\mathcal{C}$ (*completo-$\mathcal{C}$*) si además de ser difícil para la clase $\mathcal{C}$ está también en la clase $\mathcal{C}$.
- $\mathcal{C}$ es *cerrada-m* o *cerrada*-1 si es cerrada respecto a la reducibilidad $\preceq_m$ o $\preceq_1$ respectivamente.

Sea G una función total. $B \subset N$ es *recursivamente enumerable-G, r.e.-G,* si es el dominio de una función recursiva-G. Como en el caso sin oráculos se tiene las propiedades siguientes:

Proposición 5.5.6 *Si B es recursivo-G entonces B es r.e.-G.*

Proposición 5.5.7 *B es recursivo-G si y sólo si ambos B y B^c son r.e.-G.*

Proposición 5.5.8 *La clase de conjuntos r.e.-G es cerrada bajo las operaciones conjuntistas de unión e intersección.*

Proposición 5.5.9 *Los conjuntos r.e.-G son las proyecciones de las relaciones recursivas-G.*

Proposición 5.5.10 *Sea $\left\{f_n^G\right\}_n$ una enumeración de las funciones recursivas-G. Sean*

$$\begin{aligned} PP^G &= \{(n,m) | f_n^G(m)\downarrow\}, \\ DP^G &= \{n | (n,n) \in PP^G\}. \end{aligned}$$

Entonces ambos conjuntos PP^G, DP^G son r.e.-G mas no son recursivos-G.

Proposición 5.5.11 *Para cada conjunto $A \subset N$, las siguientes condiciones son equivalentes a pares:*

1. *A es r.e.-G.*
2. *$A \preceq_1 DP^G$.*

3. $A \preceq_m DP^G$.

Proposición 5.5.12 *Para cada función total* G*, las sucesiones de conjuntos* $\left\{D_n^G\right\}_n$ *y de funciones* $\left\{F_n^G\right\}_n$ *definidas simultáneamente de manera recursiva:*

$$\begin{array}{rcl} F_0^G & = & G \\ F_{n+1}^G & : & x \mapsto \begin{cases} 1 & \text{si } x \in D_n^G, \\ 0 & \text{en otro caso.} \end{cases} \end{array} \qquad \begin{array}{rcl} D_0^G & = & DP^{F_0} \\ D_{n+1}^G & = & DP^{F_{n+1}} \end{array}$$

definen una jerarquía de "irresolubilidades".

Capítulo 6

Teoremas de jerarquía

Presentaremos en este capítulo los fundamentos de las jerarquías espacial y temporal de la clase de los problemas tratables computacionalmente. Inicialmente presentamos los conceptos de *órdenes de crecimiento* de funciones reales, pues en términos de ellos nos referiremos a las complejidades de los problemas tratables. Veremos que en la clase de esos problemas no hay problemas "más difíciles" en el sentido de que dado cualquiera, siempre se puede tener un problema más difícil que ése. Veremos algunas condiciones suficientes para distinguir niveles de complejidad en las jerarquías deterministas espacial y temporal, compararemos algunos niveles de ellas y posteriormente veremos propiedades de las jerarquías no-deterministas. Finalmente, con el teorema de Borodin veremos que se puede tener dos funciones muy diversas que determinan, sin embargo, un mismo nivel en la jerarquía determinista espacial.

6.1 Ordenes de funciones

6.1.1 Definiciones básicas

Sean $f, g : R \to R$ dos funciones reales.

- *Acotamiento superior asintótico*: Escribiremos $f \underset{n\to+\infty}{=} O(g)$ si

$$\exists C > 0, n_C \in N : \;\; n \geq n_C \;\Rightarrow\; |f(n)| \leq C|g(n)|.$$

- *Dominación asintótica*: Escribiremos $f \underset{n\to+\infty}{=} o(g)$ si $\lim_{n\to+\infty} \left|\frac{f(n)}{g(n)}\right| = 0$, en otras palabras, si

$$\forall \epsilon > 0 \exists \delta > 0 : \;\; n \geq \delta \;\Rightarrow\; \left|\frac{f(n)}{g(n)}\right| < \epsilon.$$

- *Acotamiento inferior asintótico*: Escribiremos $f \underset{n\to+\infty}{=} \Omega(g)$ si

$$\exists C > 0, n_C \in N : \;\; n \geq n_C \;\Rightarrow\; |f(n)| > C|g(n)|.$$

  Consecuentemente

$$f \underset{n\to+\infty}{=} \Omega(g) \;\Leftrightarrow\; g \underset{n\to+\infty}{=} O(f)$$

De ahora en adelante, sustituiremos la notación "$\underset{n\to+\infty}{=}$" por "=".

Escribamos $f \preceq_O g$ si $f = O(g)$. Entonces "$\preceq_O$" es una relación reflexiva y transitiva.

Definamos la relación de *mismos órdenes* "$\asymp$" como sigue:

$$f \asymp g \;\Leftrightarrow\; (f \preceq_O g)\,\&\,(g \preceq_O f).$$

Entonces $\asymp$ es una relación de *equivalencia*, es decir, es reflexiva, simétrica y transitiva.

Abusando de la notación denotaremos a la clase de equivalencia de una función g por $O(g)$:

$$O(g) = \{f | f \asymp g\}.$$

$O(g)$ es el conjunto de funciones *con* el mismo *orden de crecimiento* de g.

Observación 6.1.1 *Las relaciones siguientes son inmediatas:*

1. $f = o(g) \;\Rightarrow\; f = O(g)$.
2. $O(g)$ *es cerrado bajo las operaciones de*
   - *suma de funciones, y*
   - *multiplicación de funciones por un escalar,*

   en otras palabras $O(g)$ *es un espacio vectorial.*

Ahora, si E es una función creciente y g es cualquier otra función, definimos

$$E(O(g)) = \bigcup_{h \in O(g)} O(E(h)).$$

Así pues,

$$f \in E(O(g)) \;\Leftrightarrow\; \exists C_E, C_g, n_0 \in N\,[n \geq n_0 \;\Rightarrow\; |f(n)| < C_E |E(C_g g(n))|]\,.$$

6.1.2 Algunas clases de funciones

Constantes

$O(1)$ consta de las funciones acotadas.

Funciones polinomiales

Para un polinomio f con coeficientes reales denotaremos a su *grado* por ∂f. Si f y g son dos polinomios se tiene:

$$\begin{array}{lll} f = o(g) & \Leftrightarrow & \partial f < \partial g \\ f = O(g) & \Leftrightarrow & \partial f \leq \partial g \\ f \asymp g & \Leftrightarrow & \partial f = \partial g \end{array}$$

$O(n^k)$ incluye a todos los polinomios con grado a lo sumo k.

Definimos a la clase de*funciones con crecimiento polinomial* como $Poli = \bigcup_{n \geq 0} O(n^k)$.

Como subclases importantes de *Poli* están

$$\begin{array}{lcll} Line & = & O(n) & : \text{Funciones con crecimiento } \textit{lineal}. \\ Cuad & = & O(n^2) & : \text{Funciones con crecimiento } \textit{cuadrático}. \\ Cubi & = & O(n^3) & : \text{Funciones con crecimiento } \textit{cúbico}. \end{array}$$

Funciones exponenciales

Sean $a, b > 0$. Tenemos que

$$\frac{a^x}{b^x} = \left(\frac{a}{b}\right)^x \underset{x\to+\infty}{\longrightarrow} \begin{cases} 0 & \text{si } a < b \\ 1 & \text{si } a = b \\ +\infty & \text{si } a > b \end{cases}$$

Y por tanto, $a^x = o(b^x) \Leftrightarrow a < b$. Sin embargo, como $b^x = a^{\log_a(b^x)} = a^{x\cdot\log_a(b)}$, se tiene $b^x \in a^{O(x)}$. En este sentido, es irrelevante el tamaño de la base de exponenciación, la cual puede ser 2, e o 10.

La clase de funciones *con crecimiento exponencial* es pues $Exp = \bigcup_{k\geq 0} e^{O(n^k)}$.

Por otro lado, la serie de Taylor de la función exponencial es $e^x = \sum_{n\geq 0} \frac{1}{n!}x^n$. Luego, $\forall k \in N$

$$\begin{aligned} \frac{e^x}{x^k} &= \sum_{n\geq 0} \frac{1}{n!}x^{n-k} \\ &= \sum_{m=1}^{k} \frac{1}{(m-1)!}x^{m-1-k} + \frac{1}{k!} + \sum_{m\geq 1} \frac{1}{(k+m)!}x^m \end{aligned}$$

por lo cual

$$\frac{e^x}{x^k} \underset{x\to+\infty}{\longrightarrow} 0 + \frac{1}{k!} + \infty = +\infty,$$

es decir, $x^k = o(e^x)$. En otras palabras, la función exponencial domina asintóticamente a cualquier polinomial.

Funciones logarítmicas

Sean $a, b > 0$. Tenemos que $\forall x > 0$: $\log_a(x) = \log_a(b)\log_b(x)$. Por tanto $\log_a(x) \asymp \log_b(x)$.

Así pues podemos considerar al *logaritmo natural* $x \mapsto \ln x = \log_e(x)$ como representativa de la clase de las *funciones con crecimiento logarítmico* $Log = O(\log x)$.

La función logaritmo está dominada asintóticamente por la identidad. En efecto, como

$$\frac{d}{dx}\left(\frac{\ln x}{x}\right) = \frac{1}{x^2}(1 - \ln x),$$

tenemos que el cociente $\frac{\ln x}{x}$ es estrictamente decreciente, y positivo, en el intervalo $]e, +\infty[$. En particular para la sucesión de puntos $x_n = e^n$ tenemos $\frac{\ln x_n}{x_n} = \frac{n}{e^n} \underset{x\to+\infty}{\longrightarrow} 0$. Así pues $\ln x = o(x)$.

Funciones polilogarítmicas

Finalmente consideramos la clase de funciones cuyo crecimiento es polinomial sobre el logaritmo:

$$PoliLog = \bigcup_{k\geq 0} O(\log^k x) : \text{Funciones con crecimiento } \textit{polilogarítmico}.$$

6.1.3 Límites inferiores

Recordemos aquí la definición de límites inferiores. Sea $f : R \to R$ una función real. Entonces

$$\liminf_{n\to+\infty} f(x) = y_0 \Leftrightarrow \lim_{n\to+\infty} \operatorname{Inf}_{x\geq n} f(x) = y_0,$$

es decir, si para cualquier $\epsilon > 0$ existe N_ϵ tal que

$$n > N_\epsilon \;\Rightarrow\; \left| y_0 - \operatorname*{Inf}_{x \geq n} f(x) \right| < \epsilon.$$

6.2 Clases de problemas

6.2.1 Problemas de decisión

Un *problema de decisión* consta de un subconjunto $A \subset N^m \times N^n$ y consiste en decidir cuándo una instancia de él pertenece o no a A. Es pues de la forma:

Instancia: Un punto $(\mathbf{x}, \mathbf{y}) \in N^m \times N^n$.

Solución: Una respuesta $\begin{cases} 1 & \text{si } (\mathbf{x}, \mathbf{y}) \in A \\ 0 & \text{si } (\mathbf{x}, \mathbf{y}) \notin A \end{cases}$

Un autómata que resuelve el problema de decisión es propiamente un *reconocedor* del conjunto A, o, en otras palabras, es un *comprobador* de parejas (*Instancia*, *Solución*).

6.2.2 Problemas de solución

Un *problema de solución* consta de un subconjunto $A \subset N^m \times N^n$ y consiste en localizar una solución, dada una instancia.

Instancia: Un punto $\mathbf{x} \in N^m$.

Solución: Una solución $\begin{cases} \mathbf{y} & \text{si } (\mathbf{x}, \mathbf{y}) \in A \\ \perp & \text{si } \forall \mathbf{y} \in N^n : (\mathbf{x}, \mathbf{y}) \notin A \end{cases}$

En este caso el *dominio* de A es el conjunto $\mathrm{dom}(A) = \{\mathbf{x} \in N^m | \exists \mathbf{y} \in N^n : (\mathbf{x}, \mathbf{y}) \in A\}$.

Un autómata que resuelve el problema de solución es propiamente un *calculador* de soluciones de acuerdo con la "regla" A, o un *resolvedor* del problema en el sentido estricto.

6.2.3 Comprobadores y resolvedores

Las diferencias principales en ambos tipos de problemas, de decisión y de solución, son:

Comprobación	*Resolución*
En un problema de decisión se verifica que una pareja hipotética $(\mathbf{x}, \mathbf{y}) \in N^m \times N^n$ es, en efecto, una pareja de la forma (*Instancia*, *Solución*)	En un problema de solución se construye una correspondiente *Solución* para cada *Instancia* dada.
No-determinismo	*Determinismo*
En un problema de decisión se parte de una pareja hipotética $(\mathbf{x}, \mathbf{y}) \in N^m \times N^n$. Queda indeterminada la manera en la que se obtiene $\mathbf{y}$ dada $\mathbf{x}$.	En un problema de solución se construye algorítmicamente la correspondiente *Solución* de una *Instancia* dada.

Un problema de solución puede ser visto como la *proyección* de un problema de decisión, pues

$$\mathbf{x} \in \mathrm{dom}(A) \;\Leftrightarrow\; \exists \mathbf{y} \in N^n : (\mathbf{x}, \mathbf{y}) \in A.$$

Observación 6.2.1 *Las siguientes proposiciones son inmediatas:*

1. *Sea D_A un resolvedor del problema A. Podemos construir un comprobador N_A como sigue:*
   - *Dado $(\mathbf{x}, \mathbf{y}) \in N^m \times N^n$, hacemos $\mathbf{y}_0 = D_A(\mathbf{x})$.*
   - *Si $\mathbf{y} = \mathbf{y}_0$ entonces se reconoce positivamente. Se rechaza en otro caso.*
2. *Sea N_A un comprobador del problema A. Podemos construir un resolvedor D_A como sigue:*
   - *Enumeramos el espacio de posibles soluciones $N^n = (\mathbf{y}_i)_i$.*
   - *Para cada i verificamos con N_A si acaso $(\mathbf{x}, \mathbf{y}_i) \in A$, y suspendemos esta iteración la primera vez que se tiene una respuesta positiva.*

6.3 Complejidades de tiempo y espacio

6.3.1 Dispositivos de cómputo

Consideraremos una clase $\mathcal{C}$ de *dispositivos de cómputo*, los cuales pueden ser

- máquinas de Turing,
- programas de alto nivel, digamos "C",
- funciones recursivas,
- máquinas RAM,
- etc.

Como cada dispositivo de cómputo es un *programa*, es decir, es una cadena de longitud finita sobre un alfabeto finito, tenemos que $\mathcal{C}$ es numerable $\mathcal{C} = (M_i)_{i \in N}$.

Por otro lado, las *entradas* posibles a los dispositivos en $\mathcal{C}$ son cadenas de longitud finita sobre un alfabeto de entrada (usualmente Dos=$\{0,1\}$). Por tanto las entradas también son numerables.

Para un dispositivo M y una entrada $\mathbf{x}$ hacemos

$$M(\mathbf{x}) = \begin{cases} \mathbf{y} & \text{si } M \text{ se para con } \mathbf{x} \text{ y deja como salida a } \mathbf{y}. \\ \perp & \text{si } M \text{ no se para con } \mathbf{x}. \end{cases}$$

Supondremos además que la clase $\mathcal{C}$ posee un dispositivo *universal*, es decir, que hay un $M_0 \in \mathcal{C}$ tal que

$$M_0(\lceil M \rceil \cdot \mathbf{x}) = M(\mathbf{x}), \quad \forall M, \mathbf{x},$$

donde $M \mapsto \lceil M \rceil$ es una *codificación* de dispositivos en la clase $\mathcal{C}$.

Una función $f : \mathbb{N}^m \to \mathbb{N}^n$ es *computable-$\mathcal{C}$* si $\exists M_f \in \mathcal{C}\ \forall \mathbf{x} \in N^m :\ M_f(\mathbf{x}) = f(\mathbf{x})$.

6.3.2 Tiempos y espacios

Sea $M \in \mathcal{C}$ un dispositivo de cómputo. Para una entrada $\mathbf{x} = (x_1, \ldots, x_n) \in N^n$ definimos

- *longitud* de $\mathbf{x}$, $long(\mathbf{x}) \approx \sum_{i=1}^{n} \log x_i$.
- *tiempo* de $\mathbf{x}$, $tiem_M(\mathbf{x})$ es el número de instrucciones ejecutadas hasta arribar a un estado final,

- *espacio* de $\mathbf{x}$, $espa_M(\mathbf{x})$ es el número de localidades de almacenamiento ocupadas hasta arribar a un estado final.

M determina a su vez dos funciones de *tiempo* y de *espacio*[1]:

$$\begin{array}{rcl} t_M : \mathbb{N} & \to & N \\ n & \mapsto & \mathrm{Sup}\{tiem_M(\mathbf{x}) | long(\mathbf{x}) = n\} \end{array} \qquad \text{y} \qquad \begin{array}{rcl} s_M : \mathbb{N} & \to & N \\ n & \mapsto & \mathrm{Sup}\{espa_M(\mathbf{x}) | long(\mathbf{x}) = n\}. \end{array}$$

Para una clase de dispositivos de cómputo $\mathcal{C}$ y una función $f : R \to R$ definimos las clases de problemas siguientes:

$$\begin{array}{rcl} \mathrm{DTIME}_{\mathcal{C}}(f) & = & \{A | \exists M \in \mathcal{C} : (M \text{ es resolvedor de } A) \,\&t_M(n) = O(f(n))\} \\ \mathrm{DSPACE}_{\mathcal{C}}(f) & = & \{A | \exists M \in \mathcal{C} : (M \text{ es resolvedor de } A) \,\&s_M(n) = O(f(n))\} \\ \mathrm{NTIME}_{\mathcal{C}}(f) & = & \{A | \exists M \in \mathcal{C} : (M \text{ es comprobador de } A) \,\&t_M(n) = O(f(n))\} \\ \mathrm{NSPACE}_{\mathcal{C}}(f) & = & \{A | \exists M \in \mathcal{C} : (M \text{ es comprobador de } A) \,\&s_M(n) = O(f(n))\} \end{array}$$

Teorema 6.3.1 *Para cualquier función computable f tenemos que existen sendos problemas A y B tales que $A \notin \mathrm{DTIME}(f)$, y $B \notin \mathrm{DSPACE}(f)$.*

Así pues no hay límites computables para las clases de problemas.

Demostración: Dada f, veamos tan solo la construcción de A. Sean $(M_i)_i$ y $(\mathbf{x}_i)_i$ sendas enumeraciones de los dispositivos de cómputo y de las posibles entradas. Sea

$$A = \{\mathbf{x}_i | M_i \text{ deja de reconocer a } \mathbf{x}_i \text{ en a lo sumo } f(long(\mathbf{x}_i)) \text{ pasos.}\}.$$

A puede ser efectivamente reconocido utilizando la máquina universal en $\mathcal{C}$:

1. Dado $\mathbf{x}$ se calcula i tal que $\mathbf{x}_i = \mathbf{x}$ y se calcula M_i también.
2. Se calcula $t = f(long(\mathbf{x}_i))$.
3. Se realiza el cálculo de $M_i(\mathbf{x}_i)$ llevando un recuento de la duración d de la "corrida".
4. Si M_i reconoce a $\mathbf{x}$ y $d > t$ entonces se acepta a $\mathbf{x}$. En otro caso se rechaza.

Supongamos que $A \in \mathrm{DTIME}(f)$. Entonces $\exists i_0 : (M_{i_o} \text{ reconoce a } A) \,\&\, t_{M_{i_o}} = O(f)$.

Tenemos por un lado que

$$\begin{array}{rcl} \mathbf{x}_{i_o} \in A & \Rightarrow & tiem_{M_{i_o}}(long(\mathbf{x}_{i_o})) > f(long(\mathbf{x}_{i_o})) \\ & \Rightarrow & A \notin \mathrm{DTIME}(f) \end{array}$$

y esto último contradice la suposición. Y por otro lado

$$\begin{array}{rcl} \mathbf{x}_{i_o} \notin A & \Rightarrow & M_{i_o} \text{ no reconoce a } \mathbf{x}_{i_o} \\ & \Rightarrow & \mathbf{x}_{i_o} \in A \end{array}$$

lo cual también es una contradicción. q.e.d.

[1] a la que nombraremos s por el inglés *space*, pues el símbolo e se utiliza en otros contextos

6.4 Jerarquía en espacio

Una función s se dice ser *constructible en espacio* si $\exists M \in \mathcal{C}$ tal que

$$\text{i) } t_M \asymp s \quad \text{y} \quad \text{ii) } \forall n \exists \mathbf{x} : \ n = \mathit{long}(\mathbf{x}) \& \mathit{espa}_M(\mathbf{x}) = s(n)$$

El siguiente teorema determina diferentes niveles de complejidad espacial.

Teorema 6.4.1 *Sean s_1 y s_2 dos funciones constructibles en espacio, ambas mayores que log. Si*

$$\liminf_{n \to +\infty} \frac{s_1(n)}{s_2(n)} = 0,$$

entonces $\mathrm{DSPACE}(s_1) \subset \mathrm{DSPACE}(s_2)$ *y la inclusión es propia.*

Demostración: Construiremos una máquina M tal que $M \in \mathrm{DSPACE}(s_2) - \mathrm{DSPACE}(s_1)$.

Para una entrada $\mathbf{x}$ con $n = \mathit{long}(\mathbf{x})$, M funciona como sigue:

1. M marca $s_2(n)$ casillas. Como s_2 es constructible en espacio esto se logra haciendo correr "en paralelo" a M con una máquina que construye a s_2.

   En todo momento, si M fuera a ocupar espacio fuera de lo marcado se la detiene y se suspende su corrida sin reconocimiento.

2. Calcula i tal que $\mathbf{x} = \mathbf{x}_i$ y la máquina M_i.

3. M reconoce a $\mathbf{x}$ si

   - M_i ocupa espacio s_1,
   - M realiza la simulación de M_i sobre $\mathbf{x}_i$ en espacio $s_2(n)$, y
   - M_i NO acepta a $\mathbf{x}_i$.

Claramente $M \in \mathrm{DSPACE}(s_2)$. Veamos que $M \notin \mathrm{DSPACE}(s_1)$.

En efecto, si acaso $M \in \mathrm{DSPACE}(s_1)$, entonces $\exists n : \ M = M_n \in \mathrm{DSPACE}(s_1)$. De hecho, existiría una sucesión creciente de índices $(\phi(n))_n$ tal que $M_{\phi(n)} \in \mathrm{DSPACE}(s_1)$ es equivalente a M. Ya que $\liminf_{n \to +\infty} \frac{s_1(n)}{s_2(n)} = 0$, se tiene que para cualquier constante positiva c, $\exists m : \ n > m \Rightarrow c|s_1(n)| < |s_2(n)|$. Así pues, para n suficientemente grande M podría simular en espacio s_2 a $M_{\phi(n)}$ a partir de la entrada inicial $\mathbf{x}_{\phi(n)}$. En estas condiciones se tendría $\forall n : \ (M \text{ reconoce a } \mathbf{x}_{\phi(n)} \ \Leftrightarrow \ M_{\phi(n)} \text{ reconoce a } \mathbf{x}_{\phi(n)})$, lo cual está en contradicción con la definición de M. q.e.d.

6.5 Jerarquía en tiempo

Una función t se dice ser *constructible en tiempo* si $\exists M \in \mathcal{C}$ tal que

$$\text{i) } t_M \asymp t \quad \text{y} \quad \text{ii) } \forall n \exists \mathbf{x} : \ n = \mathit{long}(\mathbf{x}) \& \mathit{tiem}_M(\mathbf{x}) = t(n)$$

El siguiente teorema determina diferentes niveles de complejidad temporal.

Teorema 6.5.1 *Sean t_1 y t_2 dos funciones constructibles en tiempo. Si*

$$\liminf_{n \to +\infty} \frac{t_1(n) \log t_1(n)}{t_2(n)} = 0,$$

entonces $\mathrm{DTIME}(t_1) \subset \mathrm{DTIME}(t_2)$ *y la inclusión es propia.*

Demostración: La demostración es similar al caso espacial. La condición de límite es más fuerte debido al siguiente

Lema 6.5.1 *Si M es una máquina de Turing de varias cintas y función de tiempo t entonces existe una máquina de Turing M_1 equivalente a ella con tan sólo dos cintas y función de tiempo $t_1 = t \log t$.*

Construiremos una máquina M tal que $M \in \text{DTIME}(t_2) - \text{DTIME}(t_1)$.

Para una entrada $\mathbf{x}$ con $n = long(\mathbf{x})$, M funciona como sigue:

1. En todo momento M corre "en paralelo" con una máquina que construye en tiempo a t_2.
2. Calcula i tal que $\mathbf{x} = \mathbf{x}_i$ y la máquina M_i.
3. M reconoce a $\mathbf{x}$ si
   - M_i corre en tiempo t_1,
   - M realiza la simulación de M_i sobre $\mathbf{x}_i$ en tiempo $t_2(n)$, y
   - M_i NO acepta a $\mathbf{x}_i$.

Claramente $M \in \text{DTIME}(t_2)$. Veamos que $M \notin \text{DTIME}(t_1)$.

En efecto, si acaso $M \in \text{DTIME}(t_1)$, entonces $\exists n : \ M = M_n \in \text{DTIME}(t_1)$. De hecho, existiría una sucesión creciente de índices $(\phi(n))_n$ tal que $M_{\phi(n)} \in \text{DTIME}(t_1)$ sería equivalente a M. Ya que $\liminf_{n \to +\infty} \dfrac{t_1(n) \log t_1(n)}{t_2(n)} = 0$, se tiene que para cualquier constante positiva c, $\exists m : \ n > m \Rightarrow c|t_1(n) \log t_1(n)| < |t_2(n)|$. Así pues, para n suficientemente grande M podría simular en tiempo t_2 a $M_{\phi(n)}$ a partir de la entrada inicial $\mathbf{x}_{\phi(n)}$. En estas condiciones se tendría $\forall n : \ (M$ reconoce a $\mathbf{x}_{\phi(n)} \ \Leftrightarrow \ M_{\phi(n)}$ reconoce a $\mathbf{x}_{\phi(n)})$, lo cual entra en contradicción con la definición de M. q.e.d.

6.6 Algunas relaciones entre clases

Las siguientes proposiciones son inmediatas:

Proposición 6.6.1 $\text{DTIME}(f(n)) \subset \text{DSPACE}(f(n))$.

En efecto, si se hacen $f(n)$ movimientos no se puede ocupar más de $f(n)$ localidades.

Proposición 6.6.2 $(L \in \text{DSPACE}(f(n)))\&(f(n) > \log n) \ \Rightarrow \ (L \in \text{DTIME}(f(n))$.

En efecto: Si $M \in \mathcal{C}$ posee q estados, utiliza un alfabeto de entrada de m símbolos y ocupa espacio $f(n)$ entonces el número de *descripciones instantáneas* (DI) es $q(n+2)f(n)m^{f(n)}$. Como $f(n) > \log n$ existe c tal que $c^{f(n)} \geq q(n+2)f(n)m^{f(n)}$. Esta es una cota para el número de DI's a generarse por un simulador de M.

Proposición 6.6.3 $\text{NTIME}(f(n)) \subset \text{DTIME}(c^{f(n)})$.

En efecto, un árbol de altura f tiene del orden de $2^{O(f)}$ nodos.

Proposición 6.6.4 (Lema de Savitch) *Se tiene la inclusión* $\text{NSPACE}(f(n)) \subset \text{DSPACE}(f^2(n))$ *siempre que f sea constructible en espacio.*

Con espacio $f(n)$ se tiene $c^{f(n)} = 2^{O(f(n))}$ DI's. Dadas dos DI's I, J una *computación legal* de longitud $k+1$ es una sucesión de DI's

$$I = I_0 \stackrel{M}{\rightarrow} I_1 \stackrel{M}{\rightarrow} \cdots \stackrel{M}{\rightarrow} I_k = J.$$

Escribamos $I \stackrel{M,2^i}{\vdash} J$ para denotar que se arriba de I a J por una computación legal de longitud 2^i. Tenemos

$$I \stackrel{M,2^i}{\vdash} J \Leftrightarrow \exists K : I \stackrel{M,2^{i-1}}{\vdash} K \& K \stackrel{M,2^{i-1}}{\vdash} J.$$

Se puede revisar esto último en espacio $f^2(n)$.

6.7 Jerarquías en clases no-deterministas

Sea $\mathcal{K}$ cualquiera de los símbolos NSPACE, DSPACE, NTIME, DTIME.

Lema 6.7.1 *Si F_1, F_2, f son constructibles entonces*

$$\mathcal{K}(F_1(n)) \subset \mathcal{K}(F_2(n)) \Rightarrow \mathcal{K}(F_1 \circ f(n)) \subset \mathcal{K}(F_2 \circ f(n)).$$

De aquí resulta

Teorema 6.7.1 *Para todo $r \geq 0$ se cumple: $\epsilon > 0 \Rightarrow$ NSPACE$(n^r) \subset$ NSPACE$(n^{r+\epsilon})$ propiamente.*

Demostración: Dados r, ϵ sean p, q enteros positivos tales que $r \leq \frac{p}{q} < \frac{p+1}{q} \leq r+\epsilon$, veamos que la inclusión NSPACE$(n^{\frac{p}{q}}) \subset$ NSPACE$(n^{\frac{p+1}{q}})$ se cumple propiamente.

Si acaso NSPACE$(n^{\frac{p+1}{q}}) \subset$ NSPACE$(n^{\frac{p}{q}})$ al componer con la función $f_i(n) = n^{(p+i)q}$ obtendríamos NSPACE$(n^{(p+1)(p+i)}) \subset$ NSPACE$(n^{p(p+i)})$.

Como $p(p+i) \leq (p+1)(p+i-1)$ tendríamos NSPACE$(n^{(p+1)(p+i)}) \subset$ NSPACE$(n^{(p+1)(p+i-1)})$.

Reiterando, para $i = p, p-1, \ldots, 1$ obtendríamos NSPACE$(n^{(p+1)(2p)}) \subset$ NSPACE$(n^{(p+1)p})$.

Por la misma razón NSPACE$(n^{(p+1)p}) \subset$ NSPACE$(n^{p \cdot p})$.

Por tanto NSPACE$(n^{(p+1)(2p)}) \subset$ NSPACE(n^{p^2}).

De acuerdo con la Prop. 6.6.4 anterior tendríamos NSPACE$(n^{p^2}) \subset$ DSPACE$((n^{p^2})^2)$.

Ahora bien, como $\liminf_{n \to +\infty} \frac{n^{2p^2}}{n^{2(p^2+p)}} = 0$ se tendría DSPACE$((n^{p^2})^2) \subset$ DSPACE$(n^{2(p^2+p)})$ propiamente.

Encadenando las inclusiones anteriores concluiríamos

$$\text{NSPACE}(n^{(p+1)(2p)}) \subset \text{DSPACE}(n^{2(p^2+p)}) \subset \text{NSPACE}(n^{(p+1)(2p)})$$

donde la primera inclusión es propia.

Esta contradicción prueba el teorema. q.e.d.

También como en el caso determinista, de manera similar al teorema 6.5.1, se tiene el siguiente teorema de separación:

Teorema 6.7.2 *Sean t_1 y t_2 dos funciones constructibles en tiempo. Si*

$$\liminf_{n \to +\infty} \frac{t_1(n) \log t_1(n)}{t_2(n)} = 0,$$

entonces NTIME$(t_1) \subset$ NTIME(t_2) *y la inclusión es propia.*

Input: A non-negative integer $n \in N$.
Output: The corresponding value $s(n)$.

{ $j := 1$;
while$\exists i \leq n$ such that $j + 1 \leq s_{M_i}(n) \leq g(j)$ **do**
$j := s_{M_i}(n)$;
$s(n) := j$
}

Figura 6.1: Construcción de la función s.

6.8 Teorema de Borodin y consecuencias

Teorema 6.8.1 (Borodin) *Para cualquier función computable g, con $g(n) \geq n$, existe una función computable s_g tal que*

$$\text{DSPACE}(s_g(n)) = \text{DSPACE}(g \circ s_g(n)).$$

En otras palabras DSPACE se colapsa entre s_g y $g \circ s_g$.

Demostración: Sea $(M_i)_i$ una enumeración de los dispositivos en $\mathcal{C}$, y s_{M_i} la función de espacio de M_i, $i \geq 0$.

Se construirá una función s tal que

$$\begin{array}{lcll} s_{M_i} & \leq & s & \text{c.t.p. (casi en todas partes), o} \\ s_{M_i} & \geq & g \circ s & \text{u.i.v. (una infinidad de veces).} \end{array}$$

Así pues ninguna función s_{M_i} puede estar "entre" s y $g \circ s$.

Construimos s mediante el algoritmo de la Figura 6.1:

Ahora, supongamos que $L \in \text{DSPACE}(g \circ s(n)) - \text{DSPACE}(s(n))$. Entonces para M_i tal que $L = L(M_i)$ tendremos $s_{M_i} \leq g \circ s$ c.t.p., y por la construcción de s necesariamente se tendrá $n \geq i \Rightarrow s_{M_i}(n) \leq s(n)$, consecuentemente $L \in \text{DSPACE}(s(n))$, lo cual es una contradicción, q.e.d.

Corolario 6.8.1 *Existe una función computable f tal que*

$$\text{DTIME}(f(n)) = \text{NTIME}(f(n)) = \text{DSPACE}(f(n)) = \text{NSPACE}(f(n)).$$

Demostración: Para cualquier función g se tiene las inclusiones siguientes

$$\begin{array}{ccc} \text{DTIME}(g(n)) & \subset & \text{DSPACE}(g(n)) \\ \cap & & \cap \\ \text{NTIME}(g(n)) & \subset & \text{NSPACE}(g(n)) \end{array}$$

Veremos que existe f tal que $\text{NSPACE}(f(n)) = \text{DTIME}(f(n))$.

Vimos que para cualquier g existe $c > 0$ tal que $\text{NSPACE}(g(n)) \subset \text{DTIME}(c^{g(n)})$, y cuantimás hemos de tener $\text{NSPACE}(g(n)) \subset \text{DTIME}(g(n)^{g(n)})$.

Ahora, para la función $G(n) = n^n$, por el análogo del Teorema de Borodin para *tiempo* tenemos que $\exists f : \text{DTIME}(f(n)) = \text{DTIME}(f(n)^{f(n)})$. f es pues la función buscada. q.e.d.

Capítulo 7

Completitud-NP

Comenzamos este capítulo con la presentación de diversas clases de problemas. Al caracterizar una clase, una noción fundamental es la de reducibilidad.

Introduciremos luego a las clases de problemas tratables en tiempo y espacio polinomiales. Bien que el determinismo y el no-determinismo coinciden en diversos niveles de la Jerarquía de Chomski, tales como el de los lenguajes regulares y el de los lenguajes recursivos, desde el punto de vista computacional, la equivalencia de estas dos nociones plantea uno de los problemas clásicos en la actualidad: decidir si la clase de problemas resolubles en tiempo polinomial coincide con la clase de problemas comprobables en tiempo polinomial, es decir, decidir si acaso P=NP. Presentamos también las nociones de reducibilidad de problemas y con ello las de dificultad y completitud de problemas en una clase dada de problemas. Presentaremos el célebre teorema de Cook que asevera que el problema SAT de decidir si acaso una forma booleana es satisfactible, es completo en la clase NP. También veremos que restricciones de estos problemas a ciertos tipos de formas booleanas continúan siendo completos-NP.

7.1 Clases de problemas

7.1.1 Reducibilidades

Recordemos algunas de las nociones introducidas en la sección 5.5.2. Ahí convinimos en que para dos conjuntos de números naturales, A, B no vacíos, $A \preceq_r B$ denota que $\chi_A \in Rec(\chi_B)$, $A \preceq_m B$ denota que existe una función computable f tal que $A = f^{-1}(B)$ y, finalmente, $A \preceq_m B$ denota que existe una función computable e inyectiva f tal que $A = f^{-1}(B)$. Las relaciones "$\preceq_r$", "$\preceq_m$" y "$\preceq_1$" son reducibilidades, es decir, son reflexivas y transitivas, y cada una es un refinamiento de la anterior.

De manera más general, si $\mathcal{F}$ es una familia de funciones, escribiremos $A \preceq_{\mathcal{F}} B$ para denotar que existe una función $f \in \mathcal{F}$ tal que $A = f^{-1}(B)$. Obviamente, "$\preceq_{\mathcal{F}}$" es reducibilidad si, por ejemplo, la familia $\mathcal{F}$ contiene a la identidad y es cerrada bajo composición.

Si "$\preceq$" es una reducibilidad, ésta induce a la relación "$\equiv$" definida como sigue:

$$A \equiv B \;\Leftrightarrow\; (A \preceq B)\&(B \preceq A).$$

Proposición 7.1.1 *"$\equiv$" es una relación de equivalencia.*

Sea $\mathcal{C}$ una clase de conjuntos y "$\preceq$" una reducibilidad.

Proposición 7.1.2 *Sea A un conjunto completo en la clase $\mathcal{C}$ y sea $B \in \mathcal{C}$ tal que $A \preceq B$. Entonces B es completo en la clase $\mathcal{C}$.*

Definamos co-$\mathcal{C} = \{A \subset N | (N - A) \in \mathcal{C}\}$. De manera inmediata, se tiene las siguientes proposiciones:

Proposición 7.1.3 *Si* co-$\mathcal{C}$ *fuera cerrado y hubiera un conjunto* A *completo para* $\mathcal{C}$ *que estuviera también en* co-$\mathcal{C}$ *entonces se ha de tener* $\mathcal{C} =$ co-$\mathcal{C}$.

En efecto, bajo las hipótesis hechas se prueba fácilmente que $\mathcal{C} \subset$ co-$\mathcal{C}$. De manera completamente simétrica se prueba la inclusión opuesta.

Proposición 7.1.4 *Si* $A \preceq_m B$ *entonces* $(N - A) \preceq_m (N - B)$. También: *Si* $A \preceq_1 B$ *entonces* $(N - A) \preceq_1 (N - B)$.

En efecto, si $f : N \to N$ es tal que $A = f^{-1}(B)$, evidentemente se tendrá también que $(N - A) = f^{-1}(N - B)$.

Proposición 7.1.5 *Si* $\mathcal{C}$ *fuera cerrada-m o cerrada-*1 *entonces también lo es* co-$\mathcal{C}$.

Esto es una consecuencia de la proposición anterior.

Proposición 7.1.6 *Sea* $\{f_n\}_n$ *una enumeración de las funciones recursivas y sea* $K = \{n | f_n(n) \downarrow\}$. *Entonces,* K *es completo-1 para la clase de conjuntos recursivamente enumerables.*

En efecto, sea A recursivamente enumerable. Hemos de probar que para alguna función recursiva e inyectiva f_0 se tiene $A = f_0^{-1}(K)$. Como A es recursivamente enumerable existe una función recursiva f_A cuyo dominio es precisamente A. Consideremos la función recursiva,

$$\begin{array}{rcl} g : N^2 & \to & N \\ (x, y) & \mapsto & g(x, y) = f_A(x) \end{array}$$

Por el Teorema de Parametrización, existe una función recursiva f_0 tal que $\forall y : \ g(x, y) = f_{f_0(x)}(y)$. Observemos que las funciones de la forma $f_{f_0(x)}, x \in N$, son todas constantes. Por tanto el predicado $\Phi(x_1, x_2) \equiv \left(f_{f_0(x_1)} = f_{f_0(x_2)}\right)$ es recursivo. Así pues, se puede modificar fácilmente a f_0 de manera que sea inyectiva si acaso no lo fuera ya.

Se cumplen además las equivalencias siguientes:

$$\begin{array}{rcccl} x \in A & \Leftrightarrow & f_A(x) \downarrow & \Leftrightarrow & \forall y : \ g(x, y) \downarrow \\ & \Leftrightarrow & g(x, f_0(x)) \downarrow & \Leftrightarrow & f_{f_0(x)}(f_0(x)) \downarrow \\ & \Leftrightarrow & f_0(x) \in K & & \end{array}$$

Proposición 7.1.7 *Si* A *es un conjunto completo-m para la clase de conjuntos recursivamente enumerables entonces su complemento* A^c *no puede ser recursivamente enumerable.*

Se tiene que la clase de los conjuntos recursivamente enumerables es cerrada-m. Luego, si A^c fuera recursivamente enumerable entonces el complemento de todo recursivamente enumerable también sería recursivamente enumerable. Evidentemente, esta última propiedad no puede ser cierta.

Proposición 7.1.8 *Dados,* $A, B \subset N$ *sea* $A \oplus B = \{2x | x \in A\} \cup \{2x + 1 | x \in B\}$. *Entonces,*

1. $A, B \preceq_m A \oplus B$.
2. $\forall C \subset N : \ A, B \preceq_m C \ \Rightarrow \ A \oplus B \preceq_m C$.

$A \oplus B$ es el conjunto que *contiene la información de ambos* A *y* B *y nada más.*

7.2 Problemas difíciles y completos en clases polinomiales

La clase de problemas *resolubles en tiempo polinomial* es $\mathrm{P} = \bigcup_{i\geq 0} \mathrm{DTIME}(n^i)$ y la clase de problemas *comprobables en tiempo polinomial* es $\mathrm{NP} = \bigcup_{i\geq 0} \mathrm{NTIME}(n^i)$. Similarmente, las clases de problemas *resolubles* en espacio polinomial es $\mathrm{PSPACE} = \bigcup_{i\geq 0} \mathrm{DSPACE}(n^i)$, en tanto que la de *comprobables* en espacio polinomial es $\mathrm{NSPACE} = \bigcup_{i\geq 0} \mathrm{NSPACE}(n^i)$.

De acuerdo con el lema de Savitch (prop. 6.6.4) se tiene dos jerarquías en PSPACE:

$$\bigcup_{i\geq 1} \mathrm{DSPACE}(\log^i n) = \mathrm{PSPACE} = \bigcup_{i\geq 1} \mathrm{NSPACE}(\log^i n).$$

Puede verse además que se tiene las contenciones siguientes

$$\mathrm{DSPACE}(\log n) \subset \mathrm{P} \subset \mathrm{NP} \subset \mathrm{PSPACE}$$

y aunque alguna de ellas es propia no se sabe cuál lo es.

En lo sucesivo, consideraremos máquinas de Turing que calculan funciones $\mathbb{N} \to \mathbb{N}$. Cada una de tales máquinas actúa a partir de descripciones instantáneas iniciales de la forma $q_0\mathbf{x}$ donde $\mathbf{x}$ es la representación en base 2 de un número $x \in \mathbb{N}$ y, en el caso de converger, finaliza con una descripción final de la forma $\mathbf{y}q_h$ donde $\mathbf{y}$ es la representación en base 2 de la imagen de x bajo la función calculada por M.

Sean pues A y B dos conjuntos en $\mathbb{N}$. Diremos que

- B se *reduce en tiempo polinomial* a A, $B \preceq_{tp} A$, si existe una máquina de Turing M de tiempo polinomial tal que $\forall x \in \mathbb{N} :\ x \in B \ \Leftrightarrow\ M(x) \in A$.
- B se *reduce en espacio-logarítmico* a A, $B \preceq_{el} A$, si existe un *traductor logarítmico*, es decir una máquina de Turing M tal que M converge en todas sus entradas, tiene una cinta de entrada, otra de salida y otras de trabajo, en la de entrada sólo se mueve en una dirección y ocupa un espacio logarítmico en las cintas que no son de entrada ni de salida, tal que $\forall x \in \mathbb{N} :\ x \in B \ \Leftrightarrow\ M(x) \in A$.

Naturalmente, se tiene que $\preceq_{tp}$ y $\preceq_{el}$ son reducibilidades, en el sentido definido en la sección 7.1.1. Se tiene las proposiciones siguientes:

Proposición 7.2.1 *Si B se reduce a A en espacio-logarítmico entonces también B se reduce a A en tiempo polinomial. Es decir,*

$$B \preceq_{el} A \ \Rightarrow\ B \preceq_{tp} A$$

En efecto, por un lado, todo traductor logarítmico ha de converger en todas sus entradas, y por otro lado, la única manera de evitar que "se ciclen" las descripciones instantáneas de cualquier máquina de Turing con espacio-logarítmico es teniendo un número polinomial de ellas en cada computación.

Proposición 7.2.2 *Si B se reduce a A en tiempo polinomial entonces*

- $A \in \mathrm{NP} \ \Rightarrow\ B \in \mathrm{NP}$.
- $A \in \mathrm{P} \ \Rightarrow\ B \in \mathrm{P}$.

Esto se debe solamente a que la composición de dos polinomios sigue siendo un polinomio.

Proposición 7.2.3 *Si B se reduce a A en espacio-logarítmico entonces*

- $A \in \mathrm{P} \;\Rightarrow\; B \in \mathrm{P}$.
- $A \in \mathrm{DSPACE}(\log^i n) \;\Rightarrow\; B \in \mathrm{DSPACE}(\log^i n)$.
- $A \in \mathrm{NSPACE}(\log^i n) \;\Rightarrow\; B \in \mathrm{NSPACE}(\log^i n)$.

Aunque la salida de un traductor logarítmico puede ser de longitud polinomial, si se tiene un tal traductor M_1 y un reconocedor en espacio logarítmico la composición de ellos se realiza en espacio logarítmico haciendo que la salida de M_1 se vaya aplicando símbolo a símbolo a la entrada de M_2.

Proposición 7.2.4 *La composición de reducciones en espacio logarítmico es de espacio logarítmico. Lo mismo vale para reducciones de tiempo polinomial.*

Sea $\mathcal{C}$ una clase de problemas, es decir, una clase de conjuntos en $\mathbb{N}$, y "$\preceq$" una reducibilidad. Sea $A \subset \mathbb{N}$ un problema.

- A es *difícil-$\mathcal{C}$, respecto a la reducibilidad "$\preceq$",* si $\forall B : (B \in \mathcal{C} \;\Rightarrow\; B \preceq A)$.
- A es *completo-$\mathcal{C}$, respecto a la reducibilidad "$\preceq$",* si es difícil-$\mathcal{C}$ y además está en la misma clase $\mathcal{C}$.

Así pues para ver que un problema dado es completo en una clase hay que probar que está en la clase y que todo otro problema en esa clase se reduce a él, respecto a la reducibilidad considerada.

Un problema es *completo*-NP o *difícil*-NP si es completo o difícil en la clase NP, con respecto a la reducción en espacio logarítmico.

Un problema es *completo*-PSPACE o *difícil*-PSPACE si es completo o difícil en la clase PSPACE, con respecto a la reducción en tiempo polinomial.

Proposición 7.2.5 *Si un problema completo-*NP *estuviera también en la clase* P *entonces necesariamente* $\mathrm{P} = \mathrm{NP}$.

7.3 Formas proposicionales y el teorema de Cook

Veremos que el problema de decidir si acaso una forma booleana es satisfactible es un problema completo-NP.

7.3.1 Formas booleanas

Utilizaremos los siguientes símbolos constantes

- **1:** Valor *verdadero*. Es la unidad de "$\wedge$": $F \wedge \mathbf{1} = \mathbf{1} \wedge F = F$.
- **0:** Valor *falso*. Es la unidad de "$\vee$": $C \vee \mathbf{0} = \mathbf{0} \vee C = C$.

Sea $\mathbf{X} = [X_j]_{j=1,\ldots,n_0}$ una lista de *variables*. Consideremos las siguientes estructuras sintácticas:

Literales Variables o negaciones de variables: L es *literal* si $\exists X \in \mathbf{X} :\; (L = X) \vee (L = \neg X)$.

Cláusulas Disyunciones de literales: Para cada $\boldsymbol{\epsilon} \in \{-1,0,1\}^{n_0}$ hacemos $\mathrm{Claus}(\boldsymbol{\epsilon}) = \bigvee_{1\le j\le n} X_j^{\downarrow\epsilon_j}$ donde

$$\forall j = 1,\ldots,n:\ X_j^{\downarrow\epsilon_j} = \begin{cases} X_j & \text{si } \epsilon_j = 1 \\ \mathbf{0} & \text{si } \epsilon_j = 0 \\ \neg X_j & \text{si } \epsilon_j = -1 \end{cases}$$

En una cláusula $\mathrm{Claus}(\boldsymbol{\epsilon})$ tenemos

$$\epsilon_j = \begin{cases} 1 & \text{si } X_j \text{ aparece en Claus}(\boldsymbol{\epsilon}), \\ -1 & \text{si } \neg X_j \text{ aparece en Claus}(\boldsymbol{\epsilon}), \\ 0 & \text{si ni } X_j \text{ ni } \neg X_j \text{ aparecen en Claus}(\boldsymbol{\epsilon}). \end{cases}$$

Frases Conjunciones de literales: Para cada $\boldsymbol{\epsilon} \in \{-1,0,1\}^{n_0}$ hacemos $\mathrm{Frase}(\boldsymbol{\epsilon}) = \bigwedge_{1\le j\le n} X_j^{\uparrow\epsilon_j}$ donde

$$\forall j = 1,\ldots,n:\ X_j^{\uparrow\epsilon_j} = \begin{cases} X_j & \text{si } \epsilon_j = 1 \\ \mathbf{1} & \text{si } \epsilon_j = 0 \\ \neg X_j & \text{si } \epsilon_j = -1 \end{cases}$$

Como en el caso de las cláusulas $\boldsymbol{\epsilon}$ indica cuáles variables aparecen en la frase, y en este caso indica a su signo de aparición.

Formas conjuntivas Conjunciones de cláusulas. Si $FC = \bigwedge_{1\le i\le m} C_i = \bigwedge_{1\le i\le m} \bigvee_{1\le j\le n} L_{ij}$ entonces representamos a FC por la matriz $FC = [L_{ij}]^C_{i,j}$.

Formas disyuntivas Disyunciones de frases. Si $FD = \bigvee_{1\le i\le m} F_i = \bigvee_{1\le i\le m} \bigwedge_{1\le j\le n} L_{ij}$ entonces representamos a FD por la matriz $FD = [L_{ij}]^D_{i,j}$.

Formas proposicionales Puesto recursivamente,

- $\mathbf{0}, \mathbf{1}$ son formas proposicionales.
- Las variables son formas proposicionales.
- Las negaciones de formas proposicionales son también formas proposicionales.
- Las conjunciones y las disyunciones de formas proposicionales son también formas proposicionales.

Sea $\mathrm{Dos} = \{0,1\}$ el conjunto de los dos valores de verdad booleanos. Toda forma proposicional F determina una función $F : \mathrm{Dos}^{n_0} \to \mathrm{Dos}$. Para cada $\mathbf{x} \in \mathrm{Dos}^{n_0}$ $F(\mathbf{x})$ es el valor de verdad que asume F bajo la asignación $\mathbf{x}$. Dos formas son *equivalentes* si definen a una misma función.

Confundiremos voluntariamente a una forma proposicional con la función $\mathrm{Dos}^{n_0} \to \mathrm{Dos}$ que define.

Una forma proposicional F es

- *satisfactible* si hay una asignación que la satisface, es decir, si $F \neq \mathbf{0}$,
- *tautología* si es satisfecha por todas las posibles asignaciones, es decir, si $F = \mathbf{1}$,
- *refutable* si hay una asignación que la refuta, es decir, si $F \neq \mathbf{1}$,
- *insatisfactible* si es refutada por todas las posibles asignaciones, es decir, si $F = \mathbf{0}$.

Símbolo	*Código*	*Símbolo*	*Código*
(	1	**0**	110
)	10	**1**	111
$\neg$	11	x	1000
$\wedge$	100	0	1001
$\vee$	101	1	1010

Tabla 7.1: Codificación de símbolos.

7.3.2 El primer problema completo-NP

Problema SAT Consiste en decidir si acaso una forma proposicional dada es satisfactible.

Formalmente el problema se especifica como sigue:

Instancia: Una forma proposicional F.

Solución: $\begin{cases} 1 & \text{si } F \text{ es satisfactible.} \\ 0 & \text{si } F \text{ es insatisfactible.} \end{cases}$

Toda instancia $F(X_1, \ldots, X_{n_0})$ de SAT puede verse como una palabra sobre el alfabeto

$$\mathit{AlfProp}_{n_0} = \{(,), \neg, \wedge, \vee, \mathbf{0}, \mathbf{1}\} \cup \mathbf{X},$$

de $n_0 + 7$ símbolos. Si la longitud de esta palabra es n tenemos que

1. hay a lo sumo $\lceil \frac{n}{2} \rceil$ apariciones de variables,
2. la cadena se codifica como una palabra de longitud $O(n \log n)$ sobre el alfabeto $\{0, 1\}$.

   En efecto,

   (a) a cada variable X_j la codificamos por la palabra formada por un símbolo x seguido de la representación de j en base 2. Digamos, $X_5 \leftrightarrow x101$.

   (b) codifiquemos a cada uno de los símbolos en uso actual como se indica en la Tabla 7.1.

   (c) cada cadena en el lenguaje proposicional se transcribe como una cadena de cadenas de 0's y 1's. V.gr,:

   $$\neg(X_1 \wedge X_5) \leftrightarrow [11, 1, 1000, 1010, 100, 1000, 1010, 1001, 1010].$$

   (d) A una cadena de cadenas de 0's y 1's. la representamos, mediante una sola cadena de 0's y 1's, cambiando cada 1 por 10, cada 0 por 0 mismo y cada "," por 11. V.gr,:

   $$[11, 1, 1000, 1010, 100, 1000, 1010, 1001, 1010]$$
   $$\updownarrow$$
   $$1010\ 11\ 10\ 11\ 10000\ 11\ 100100\ 11\ 100011\ 10000\ 11\ 100100\ 11\ 100010\ 11\ 100100$$

3. Consecuentemente, respecto a reducciones en espacio logarítmico es irrelevante utilizar el alfabeto $\mathit{AlfProp}_{n_0}$ o el alfabeto $\{0, 1\}$ pues

   $$\log(n \log n) \leq 2 \log n.$$

   Lo mismo vale respecto a reducciones polinomiales.

Teorema 7.3.1 (Cook, 1971) SAT *es completo*-NP.

Demostración: Veamos primeramente que SAT $\in$ NP.

Dada una instancia $F = F(X_1, \ldots, X_{n_0})$, se genera una asignación $\boldsymbol{\epsilon} \in \text{Dos}^n$ de manera no determinista. La evaluación de F sobre $\boldsymbol{\epsilon}$ requiere un tiempo proporcional al número de conectivos que aparecen en F el cual está limitado por la propia longitud de F.

Veamos ahora que SAT es difícil-NP.

Para esto, hay que ver que cualquier problema en NP se reduce a SAT, para lo cual veremos que para cualquier máquina de Turing no-determinista M, acotada en tiempo por un polinomio $p(n)$, podemos construir un traductor logarítmico

$$\begin{array}{rcl} T_M : \{\text{Entradas de } M\} & \rightarrow & \{\text{Formas proposicionales}\} \\ \mathbf{x} & \mapsto & \phi_{\mathbf{x}} \end{array}$$

de manera que $\forall \mathbf{x} :\ M(\mathbf{x}) \downarrow \Leftrightarrow\ \text{SAT}(\phi_{\mathbf{x}}) = \textit{Sí}$.

Consideremos una computación terminal en M, correspondiente a una entrada $\mathbf{x}$ de longitud n. Esta es una sucesión finita de descripciones instantáneas cuya longitud es $O(p(n))$. Escribámosla $\mathit{Comp}(\mathbf{x}) = \bullet di_0 \bullet di_1 \cdots \bullet di_{p(n)}$.

Cada $di_i = [\mathbf{x}_i q_i a_i \mathbf{y}_i]$ involucra del orden de $p(n)$ símbolos. De hecho, podemos confundir a la pareja $q_i a_i$ como un solo símbolo (nuevo) $q_i a_i m_i$, donde m_i indica al movimiento que hace M estando en el estado q_i y encontrando el símbolo a_i. Así, se puede escribir la computación $\mathit{Comp}(\mathbf{x})$ como una palabra de longitud $(p(n)+1)^2$ sobre un cierto alfabeto A'.

Consideremos el conjunto de variables proposicionales $VP = \left\{X_{ia} | 0 \leq i \leq (p(n)+1)^2 - 1, a \in A'\right\}$. La intención de cada una de estas variables es

$$X_{ia} \text{ es verdadero} : \text{si el símbolo } a \text{ aparece en la posición } i \text{ de } \mathit{Comp}(\mathbf{x}).$$

La construcción de $\phi_{\mathbf{x}} \in \mathit{FormProp}(VP)$ se hará teniendo en mente lo siguiente:

1. Para cada i existe un único a tal que X_{ia} es verdadero.
2. La descripción inicial di_0 es de la forma $q_0\mathbf{x}$.
3. La última descripción $di_{p(n)}$ es terminal.
4. Cada descripción di_{i+1} se sigue de su anterior di_i según las transiciones de la máquina M.

De acuerdo con el punto 1 anterior definamos

$$\phi_{1,\mathbf{x}} = \bigwedge_i \left[\bigvee_a \left(X_{ia} \wedge \bigwedge_{b \neq a} \neg X_{ib} \right) \right]. \tag{7.1}$$

De acuerdo con el punto 2 anterior consideremos lo siguiente:

- La descripción inicial di_0, como cualquier otra, va entre dos "$\bullet$" 's. Sea

$$\phi_{2,1,\mathbf{x}} = X_{0\bullet} \wedge X_{(p(n)+1),\bullet}. \tag{7.2}$$

- El primer símbolo de di_0 corresponde al símbolo compuesto formado por el estado inicial q_0 de M y el símbolo inicial x_1 de $\mathbf{x}$. Sea

$$\phi_{2,2,\mathbf{x}} = \bigvee_{[q_0 x_1 m] \in \text{Transiciones en } M} X_{1[q_0 x_1 m]}. \tag{7.3}$$

- Los siguientes $n-1$ símbolos de di_0 corresponden a los símbolos x_i que conforman a $\mathbf{x}$. Sea

$$\phi_{2,3,\mathbf{x}} = \bigwedge_{i=2}^{n} X_{ix_i}. \tag{7.4}$$

- El resto de di_0 se llena con blancos. Sea

$$\phi_{2,4,\mathbf{x}} = \bigwedge_{i=n+1}^{p(n)} X_{i\square}. \tag{7.5}$$

Tomemos pues a la conjunción de las fórmulas en las ecuaciones 7.2-7.5:

$$\phi_{2,\mathbf{x}} = \phi_{2,1,\mathbf{x}} \wedge \phi_{2,2,\mathbf{x}} \wedge \phi_{2,3,\mathbf{x}} \wedge \phi_{2,4,\mathbf{x}}. \tag{7.6}$$

De acuerdo con el punto 3 anterior definamos

$$\phi_{3,\mathbf{x}} = \bigvee_{i=p(n)(p(n)+1)+1}^{(p(n)+1)^2-1} \left[\bigvee_{\substack{q_F \in \text{EdosFinales} \\ [q_F a m] \in \text{Transiciones}}} X_{i[q_F a m]} \right]. \tag{7.7}$$

Finalmente, para expresar la necesidad de que dos descripciones contiguas di_i y di_{i+1} se ajustan a las transiciones de M, observamos que tales dos descripciones han de coincidir en todas partes salvo en las posiciones vecinas a los símbolos compuestos correspondientes a estados. Definamos entonces un predicado $\Phi(a_1, a_2, a_3, a_4; j)$ tal que

$$\Phi(a_1, a_2, a_3, a_4; j) \quad \Leftrightarrow \quad \begin{pmatrix} \text{en la posición } j \text{ el símbolo } a_4 \text{ aparece en } di_{i+1} \text{ y la cadena } a_1a_2a_3 \\ \text{aparece en } di_i \text{ desde la posición } j-1, \text{ es decir} \\ (di_i = \mathbf{u}a_1a_2a_3\mathbf{v}) \,\&\, (di_{i+1} = \mathbf{u}c_1a_4c_2\mathbf{v}) \end{pmatrix}$$
$$\vee \ ((a_2 = \bullet) \wedge (a_4 = \bullet))$$

Con Φ podemos simular efectivamente las transiciones de M.

Sea entonces

$$\phi_{4,\mathbf{x}} = \bigwedge_{j=1}^{p(n)(p(n)+1)} \bigvee_{\Phi(a_1,a_2,a_3,a_4;j)} \left[X_{(j-1),a_1} \wedge X_{(j),a_2} \wedge X_{(j+1),a_3} \wedge X_{(p(n)+j),a_4} \right]. \tag{7.8}$$

La forma proposicional a construir es entonces la conjunción de las fórmulas 7.1, 7.6, 7.7 y 7.8:

$$\phi_{\mathbf{x}} = \phi_{1,\mathbf{x}} \wedge \phi_{2,\mathbf{x}} \wedge \phi_{3,\mathbf{x}} \wedge \phi_{4,\mathbf{x}}. \tag{7.9}$$

De su misma construcción tenemos que toda computación terminal, con inicio en $\mathbf{x}$, determina una asignación que satisface a la fórmula $\phi_{\mathbf{x}}$ y, viceversa, toda asignación que satisfaga a esta fórmula determina también a una computación terminal a partir de $\mathbf{x}$.

7.4 Otros problemas completos-NP en formas proposicionales

Problema SAT-FNC Consiste en decidir si acaso una forma proposicional dada como una forma conjuntiva es satisfactible.

Formalmente el problema se especifica como sigue:

Instancia: Una forma conjuntiva F.

Solución: $\begin{cases} 1 & \text{si } F \text{ es satisfactible.} \\ 0 & \text{si } F \text{ es insatisfactible.} \end{cases}$

Proposición 7.4.1 SAT-FNC *es completo-*NP*.*

Para esto veamos que SAT se reduce a SAT-FNC.

Lema 7.4.1 *Existe un traductor logarítmico*

$$\begin{array}{rcl} FC: \{\text{Formas proposicionales}\} & \rightarrow & \{\text{Formas conjuntivas}\} \\ \phi & \mapsto & FC(\phi) \end{array}$$

de manera que $\forall \phi$*:* ϕ *es satisfactible cuando y sólo cuando* $FC(\phi)$ *también lo sea.*

De hecho, tendremos que una asignación $\boldsymbol{\epsilon}$ satisface a ϕ si y sólo si una extensión de $\boldsymbol{\epsilon}$ satisface a $FC(\phi)$.

Demostración: Presentamos a grandes rasgos reglas de transformación para calcular $FC(\phi)$ a partir de ϕ.

1. *Aplicar negaciones al nivel más bajo*:

$$\begin{array}{rcl} \neg(\phi \wedge \psi) & \rightsquigarrow & \neg(\phi) \vee \neg(\psi) \\ \neg(\phi \vee \psi) & \rightsquigarrow & \neg(\phi) \wedge \neg(\psi) \\ \neg\neg\phi & \rightsquigarrow & \phi \end{array}$$

   Reiterando las dos primeras reglas y luego aplicando la tercera en cada variable, toda fórmula proposicional se transforma en una expresión formada a partir de literales utilizando conjunciones y disyunciones. Llamemos a éstas *formas y/o*.

   Las primeras dos reglas se realizan mediante un procedimiento similar al reconocimiento de cadenas equilibradas de paréntesis. Utilizando contadores para referirse a las posiciones de los paréntesis se las puede realizar en espacio logarítmico.

   La tercera regla se aplica directamente mediante un revisor de paridades de cadenas de "$\neg$" 's consecutivos.

2. *Cálculo de formas conjuntivas*: Definimos el operador Φ : {Formas y/o} $\rightarrow$ {Formas conjuntivas} de manera inductiva como sigue:

$$\begin{array}{rcl} \phi \text{ literal} & \Rightarrow & \Phi(\phi) = \phi \\ \left.\begin{array}{l} \phi = \phi_1 \wedge \phi_2, \\ \Phi(\phi_1) = \bigwedge_{i_1} F_{1,i_1}, \\ \Phi(\phi_2) = \bigwedge_{i_2} F_{2,i_2} \end{array}\right\} & \Rightarrow & \Phi(\phi) = \displaystyle\bigwedge_{i_1} F_{1,i_1} \wedge \bigwedge_{i_2} F_{2,i_2} \\ \left.\begin{array}{l} \phi = \phi_1 \vee \phi_2, \\ \Phi(\phi_1) = \bigwedge_{i_1} F_{1,i_1}, \\ \Phi(\phi_2) = \bigwedge_{i_2} F_{2,i_2} \end{array}\right\} & \Rightarrow & \Phi(\phi) = \displaystyle\bigwedge_{i_1} [Y \vee F_{1,i_1}] \wedge \bigwedge_{i_2} [\neg Y \vee F_{2,i_2}] \end{array}$$

   donde Y, en la última relación, es una nueva variable.

   Estas transformaciones son también susceptibles de realizarse en espacio logarítmico.

Una *forma conjuntiva-3, FC-3,* es una forma conjuntiva tal que toda cláusula en ella consta de exactamente 3 literales.

Problema 3-SAT Consiste en decidir si acaso una FC-3 dada es satisfactible.

Formalmente el problema se especifica como sigue:

Instancia: Una FC-3 F.

Solución: $\begin{cases} 1 & \text{si } F \text{ es satisfactible.} \\ 0 & \text{si } F \text{ es insatisfactible.} \end{cases}$

Proposición 7.4.2 3-SAT *es completo*-NP.

Para esto veamos que SAT-FNC se reduce a 3-SAT.

Lema 7.4.2 *Existe un traductor logarítmico*

$$\begin{array}{rcl} FC\colon \{\text{Formas conjuntivas}\} & \to & \{\text{FC-3's}\} \\ \phi & \mapsto & FC(\phi) \end{array}$$

de manera que $\forall \phi$*:* ϕ *es satisfactible cuando y sólo cuando* $FC(\phi)$ *también lo sea.*

De hecho, tendremos que una asignación $\boldsymbol{\epsilon}$ satisface a ϕ si y sólo si una extensión de $\boldsymbol{\epsilon}$ satisface a $FC(\phi)$.

Demostración: Si $Fr = L_1 \vee L_2 \vee L_3$ es una frase 3 literales hagamos $FC(Fr) = Fr$.

Si $Fr = L_1 \vee L_2 \vee \cdots \vee L_k$ es una frase con $k > 3$ literales consideremos $k-3$ nuevas variables $Y_1, \ldots, Y_{k-3}$ y hagamos

$$FC(Fr) = \begin{cases} (\ L_1 \ \vee\ L_2 \ \vee\ Y_1\ ) \wedge \\ (\ \neg Y_1 \ \vee\ L_3 \ \vee\ Y_2\ ) \wedge \\ (\ \neg Y_2 \ \vee\ L_4 \ \vee\ Y_3\ ) \wedge \\ \qquad\qquad \vdots \\ (\ \neg Y_{i-1} \ \vee\ L_{i+1} \ \vee\ Y_i\ ) \wedge \\ \qquad\qquad \vdots \\ (\ \neg Y_{k-3} \ \vee\ L_{k-1} \ \vee\ L_k\ ) \end{cases}$$

Es claro que una asignación satisface Fr si y sólo si una extensión de ella satisface $FC(Fr)$.

Sea ahora $\phi = \bigwedge_i Fr_i$ una FC. Podemos suponer que toda cláusula en ϕ consta de al menos tres literales (si no fuera así podemos repetir una literal pues la disyunción es una operación equipotente). Hagamos $FC(\phi) = \bigwedge_i FC(Fr_i)$. Entonces $FC(\phi)$ es del tipo 3-FNC y será satisfactible si y sólo si ϕ lo es.

Capítulo 8

Algunos problemas principales completos-NP

8.1 Programación lineal

Un *funcional lineal* en $\mathbb{R}^n$ está dado por un vector $\mathbf{c} \in \mathbb{R}^n$, mediante la aplicación $\mathbf{x} \mapsto \mathbf{c}^T\mathbf{x}$. El *problema de la programación lineal entera* consiste en decidir si existe un vector con coordenadas enteras en un poliedro determinado por un conjunto de desigualdades lineales, con coeficientes enteros, que hace tomar a un funcional lineal un valor suficientemente grande. Formalmente:

Instancia: Una matriz $\mathbf{A} \in \mathbb{Z}^{m\times n}$, un vector de coeficientes "independientes" $\mathbf{d} \in \mathbb{Z}^m$, los coeficientes de un funcional $\mathbf{c} \in \mathbb{Z}^n$ y un valor "grande" $v \in Z$.

Solución: $\begin{cases} 1 & \text{si } \exists \mathbf{x} \in \mathbb{Z}^n : (\mathbf{A}\mathbf{x} \leq \mathbf{d}) \wedge (\mathbf{c}^T\mathbf{x} \geq v), \\ 0 & \text{en otro caso.} \end{cases}$

8.2 Combinatoria

8.2.1 Atercetamientos

Si consideramos un conjunto M de parejas matrimoniales, entonces las buenas costumbres dictan que

- $M \subset \mathit{Hombres} \times \mathit{Mujeres}$,
- cada una de las proyecciones

$$\begin{array}{rcccc} h : M \to & \mathit{Hombres} & , & (x,y) & \mapsto & x \\ m : M \to & \mathit{Mujeres} & , & (x,y) & \mapsto & y \end{array}$$

  es inyectiva.

  Es decir, vale el sano principio: "cada quién con su cada cuál".

Esta idea de apareamientos matrimoniales se generaliza a tercetas.

Sean X, Y, Z tres conjuntos, ajenos a pares. Un *atercetamiento (matching)* es un conjunto $M \subset X \times Y \times Z$ tal que cada una de las funciones proyecciones, restringida a M, es inyectiva.

En otras palabras, en todo atercetamiento M, rige la implicación:

$$\left.\begin{array}{l}(x_1,y_1,z_1),(x_2,y_2,z_2)\in M\\(x_1,y_1,z_1)\neq(x_2,y_2,z_2)\end{array}\right\}\Rightarrow\left\{\begin{array}{l}(x_1\neq x_2)\wedge\\(y_1\neq y_2)\wedge\\(z_1\neq z_2)\end{array}\right.$$

Atercetamientos (3DM):

Este problema, llamado en inglés *3-dimensional matching*, se formaliza como sigue,

Instancia: Un subconjunto

$$M\subset X\times Y\times Z,$$

donde X,Y,Z son tres conjuntos, ajenos a pares, de una misma cardinalidad, digamos n.

Solución: $\left\{\begin{array}{ll}1 & \text{si hay un atercetamiento } M'\subset M \text{ con } n \text{ tercetas,}\\0 & \text{en otro caso.}\end{array}\right.$

Naturalmente, más que planteado como un problema de decisión 3DM se plantea como uno de solución. Así que en caso de que hubiera un tal atercetamiento, habría que localizarlo explícitamente.

Proposición: 3DM es completo-NP.

Demostración: Que este problema esté en NP es fácil de ver:

1. Se conjetura un posible atercetamiento M' incluído en M, con n tercetas.
2. Se revisa que cada una de las proyecciones cubra a los conjuntos "coordenadas".

Para probar que es difícil-NP, veamos que 3SAT se reduce a 3DM.

Dados

- $\mathbf{X}=\{X_1,\ldots,X_n\}$: conjunto de n variables proposicionales,
- $\mathbf{C}=\{c_1,\ldots,c_m\}$: conjunto de m cláusulas de 3 literales sobre $\mathbf{X}$,

construiremos una instancia de 3DM de manera que ésta dé una instancia con respuesta positiva en 3DM si y sólo si la forma conjuntiva $\mathbf{C}$ es satisfactible. En tal caso, un atercetamiento en la instancia construída ha de determinar una asignación de valores de verdad a las variables en $\mathbf{X}$ que satisfaga a $\mathbf{C}$.

Definamos

- $X=\{x_{ij},\bar{x}_{ij}|1\leq i\leq m,1\leq j\leq n\}$: Tiene $2nm$ elementos.
- $Y=A_1\cup S_1\cup G_1$ donde
  - $A_1=\{a_{1ij}|1\leq i\leq m,1\leq j\leq n\}$: Tiene nm elementos.
  - $S_1=\{s_{1i}|1\leq i\leq m\}$: Tiene m elementos.
  - $G_1=\{g_{1k}|1\leq k\leq m(n-1)\}$: Tiene $m(n-1)$ elementos.
- $Z=A_2\cup S_2\cup G_2$ donde
  - $A_2=\{a_{2ij}|1\leq i\leq m,1\leq j\leq n\}$: Tiene nm elementos.
  - $S_2=\{s_{2i}|1\leq i\leq m\}$: Tiene m elementos.
  - $G_2=\{g_{2k}|1\leq k\leq m(n-1)\}$: Tiene $m(n-1)$ elementos.

Consecuentemente, tanto Y como Z poseen $2nm$ elementos cada uno.

- *Tercetas de asignación*: Para cada variable X_j construímos los conjuntos de tercetas

$$\begin{aligned} V_j^{verda} &= \{[\bar{x}_{ij}, a_{1ij}, a_{2ij}] | 1 \leq i \leq m\} \\ V_j^{falso} &= \{[x_{ij}, a_{1,i+1,j}, a_{2ij}] | 1 \leq i < m\} \cup \{[x_{mj}, a_{1,1,j}, a_{2mj}]\} \\ V_j &= V_j^{verda} \cup V_j^{falso} \end{aligned}$$

El propósito de esta definición es que todo atercetamiento M' con $2nm$ tercetas debe satisfacer las condiciones siguientes:

$$\begin{aligned} (1 \quad &\Rightarrow) \quad M' \cap V_j \neq \emptyset \\ M' \cap V_j^{verda} \neq \emptyset \quad &\Rightarrow \quad M' \cap V_j^{verda} = V_j^{verda} \\ M' \cap V_j^{falso} \neq \emptyset \quad &\Rightarrow \quad M' \cap V_j^{falso} = V_j^{falso} \end{aligned}$$

así pues, un tal atercetamiento fuerza de manera natural un valor de verdad en la variable X_j.

- *Tercetas de satisfacción*: Para cada cláusula c_i construímos el conjunto

$$C_i = \{[x_{ij}, s_{1ij}, s_{2ij}] | X_j \in c_i\} \cup \left\{[\bar{x}_{ij}, s_{1ij}, s_{2ij}] | \bar{X}_j \in c_i\right\}.$$

- *Tercetas de relleno*: Construímos el conjunto

$$G = \{[x_{ij}, g_{1ij}, g_{2ij}], [\bar{x}_{ij}, s_{1ij}, s_{2ij}]\}_{\substack{1 \leq k \leq m(n-1) \\ 1 \leq i \leq m \\ 1 \leq j \leq n}}.$$

- *Instancia para 3DM*:

$$M = \left(\bigcup_{j=1}^{n} V_j\right) \cup \left(\bigcup_{i=1}^{m} C_i\right) \cup G.$$

Consecuentemente, $\text{card}(M) = 2mn + 3m + 2m^2n(n-1)$.

Soluciones de 3SAT corresponden a soluciones de 3DM: Supongamos que la asignación $\boldsymbol{\epsilon} : X_j \mapsto \epsilon_j$ satisface a la forma conjuntiva C. Consideremos las siguientes tercetas:

- $M_{10} = \bigcup_{\epsilon_j=0} V_j^{falso}$: Esto da $n_f \cdot m$ tercetas, donde n_f es el número de 0's en la asignación $\boldsymbol{\epsilon}$.

- $M_{11} = \bigcup_{\epsilon_j=1} V_j^{verda}$: Esto da $n_v \cdot m$ tercetas, donde n_v es el número de 1's en la asignación $\boldsymbol{\epsilon}$. Evidentemente $(n_f + n_v) \cdot m = n \cdot m$.

- Para cada cláusula c_i elijamos una literal $z_i \in \mathbf{X} \cup \bar{\mathbf{X}}$ que haga que c_i sea verdadera, es decir tal que $z_i(\epsilon_{z_i}) = 1$. Tal literal existe porque la asignación satisface a todas las cláusulas. Sea entonces

$$M_2 = \bigcup_{i=1}^{m} \{[z_i, s_{1i}, s_{2i}]\}.$$

Este conjunto consta de m tercetas.

- M_3 es un subconjunto de tercetas de la forma $[y, g_{1k}, g_{2k}]$ donde y varía sobre todas las literales que no hayan aparecido en los conjuntos de arriba, de las cuales hay precisamente

$$2nm - (nm + m) = (n-1)m$$

y los g's varían sobre todas sus posibilidades (la formación de apareamientos buenos entre parejas de g's es trivial).

Se tiene que

$$M' = M_{10} \cup M_{11} \cup M_2 \cup M_3$$

es un atercetamiento de $2nm$ tercetas incluído en la instancia M.

Soluciones de 3DM corresponden a soluciones de 3SAT: Sea M' un atercetamiento en M con $2nm$ elementos. Por la construcción de las tercetas, necesariamente

- A lo más $(n-1)m$ tercetas de M' son del tipo M_3,
- a lo más m tercetas de M' son del tipo M_2, y
- a lo más nm tercetas de M' son del tipo M_1.

Puesto que M' contiene $2nm$ tercetas las anteriores cotas se alcanzan efectivamente.

Las del tipo M_1 son de la forma

$$\bigcup_{j \in J_0} V_j^{falso} \cup \bigcup_{j \in J_1} V_j^{verda},$$

donde $J_0 \cup J_1 = [1, n]$.

Pues bien la asignación

$$X_j \mapsto \begin{cases} 1 & \text{si } j \in J_1, \\ 0 & \text{si } j \in J_0. \end{cases}$$

satisface a la instancia de 3SAT. En cada cláusula la literal que se satisface es la que aparece en la correspondiente terceta del tipo M_2 en M'.

8.2.2 Partición

Consideremos el problema siguiente

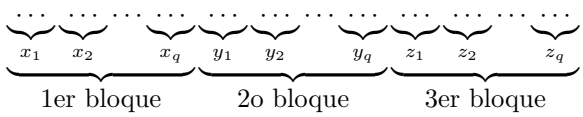
Instancia: Una sucesión $\mathbf{s} = [s_1, \ldots, s_n] \in \mathbb{N}^n$.

Solución: $\begin{cases} 1 & \text{si existe } I \subset [1,n] \text{ tal que } \sum_{i \in I} s_i = \sum_{i \notin I} s_i, \\ 0 & \text{en otro caso.} \end{cases}$

Es claro que este problema PARTITION está en NP. Para ver que es difícil-NP veamos que 3DM se reduce a él.

Sea $M = [m_1, \ldots, m_k] \subset X \times Y \times Z$ una instancia de 3DM, con

$$\begin{aligned} X &= [x_1, \ldots, x_q] \\ Y &= [y_1, \ldots, y_q] \\ Z &= [z_1, \ldots, z_q] \\ k &\geq q \end{aligned}$$



$[\underbrace{\cdots} = p$ bits.]
El número s_i tiene 1's en las posiciones más a la derecha de los segmentos de p bits etiquetados con $x_{f(i)}$, $y_{g(i)}$ y $z_{h(i)}$, y tiene 0's en las demás posiciones.

Figura 8.1: Construcción de los sumandos en $\mathbf{s}$.

Construiremos una instancia $\mathbf{s} = [s_1, \ldots, s_n] \in \mathbb{N}^n$ de PARTITION tal que

$$\left[\exists I \subset [1,n] : \sum_{i\in I} s_i = \sum_{i\notin I} s_i\right] \Leftrightarrow \exists M' \subset M : M' \text{ es un atercetamiento de } q \text{ tercetas.}$$

Consideremos pues los siguientes conceptos:

$$\begin{aligned} n = k+2 &: \text{número de sumandos en la instancia } \mathbf{s},\\ p = \lceil \log_2(k+1) \rceil &: \text{tamaño básico de bloques de "bits" para especificar cada sumando en } \mathbf{s},\\ f,g,h : [1,k] \to [1,q] &: \text{funciones que dan las proyecciones de las tercetas en } M, \text{ i.e.} \\ & \qquad \forall i \le k : m_i = [x_{f(i)}, y_{g(i)}, z_{h(i)}]. \end{aligned}$$

Para cada $i = 1, \ldots, k$ sea

$$s_i = (2^p)^{(3q-f(i))} + (2^p)^{(2q-g(i))} + (2^p)^{(q-h(i))}.$$

Cada s_i está determinado por la terceta m_i, así, escribiremos también $s(m_i) = s_i$.

Tenemos entonces que cada sumando de la sucesión $\mathbf{s}$ se escribe con exactamente $3pq$ bits.

En la Figura 8.1 presentamos diagramáticamente esta construcción.

Sea B el número con 1 en las posición más a la derecha de cada segmento de p bits,

$$B = \sum_{i=0}^{3q-1} (2^p)^i.$$

Entonces, para todo subconjunto de tercetas $M' \subset M$ se cumple que

$$\sum_{m\in M'} s(m) = B \Leftrightarrow M' \text{ es un atercetamiento.}$$

Finalmente definamos

$$s_{k+1} = 2\left(\textstyle\sum_{i=1}^k s_i\right) - B \quad , \quad s_{k+2} = \left(\textstyle\sum_{i=1}^k s_i\right) + B$$

Tenemos que $\displaystyle\sum_{i=1}^n s_i = 4\left(\textstyle\sum_{i=1}^k s_i\right)$.

Por tanto, si $I \subset [1,n]$ es tal que $\sum_{i\in I} s_i = \sum_{i\notin I} s_i$ entonces ambas sumas deben igualar el valor $2\left(\sum_{i=1}^{k} s_i\right)$. Entonces, los sumandos s_{k+1} y s_{k+2} deben estar en sumas distintas. Sus diferencias determinan pues un atercetamiento M' en M.

Resumiendo: PARTITION es un problema completo-NP.

8.3 Problemas en gráficas

Una *gráfica* es un sistema $G = (V, A)$ donde

- $V = \{v_1, \ldots, v_n\}$ es un conjunto de *vértices* o *nodos*, y
- $A \subset V^{(2)} = \{a \subset V | \text{card}(a) = 2\}$ es un conjunto de *aristas* (sin dirección).

Una gráfica se dice ser

- un *clan* o *completa* si contiene todas las $\binom{n}{2} = \frac{n(n-1)}{2}$ aristas posibles, y en tal caso se le denota como K_n,
- *vacía* o *independiente* si su conjunto de aristas es vacío, y en tal caso la denotaremos por I_n.

El *complemento*, o gráfica *complementaria*, de una gráfica $G = (V, A)$ es $G^c = (V, V^{(2)} - A)$, es decir

$$\forall v_1 \neq v_2 : \{v_1, v_2\} \text{ arista en } G^c \Leftrightarrow \{v_1, v_2\} \text{ NO es arista en } G.$$

Así, por ejemplo, K_n e I_n son gráficas complementarias una de otra.

Recordamos también los conceptos siguientes:

- un *recubrimiento por vértices* es un subconjunto $U \subset V$ de vértices tal que toda arista tiene al menos un extremo en él, es decir, tal que
$$\forall a \in A : \ a \cap U \neq \emptyset.$$
- un *recubrimiento por aristas* es un subconjunto $B \subset A$ de aristas tal que todo vértice está en al menos una de tales aristas, es decir, tal que
$$\forall v \in V \exists a \in B : v \in a.$$

Consideremos los siguientes problemas:

Recubrimiento por vértices (VC): Este problema (*vertex cover*) se describe como sigue:

Instancia: Una gráfica $G = (V, A)$ y un número $n \leq \text{card}(V)$.

Solución: $\begin{cases} 1 & \text{si existe un recubrimiento por vértices con } n \text{ vértices,} \\ 0 & \text{en otro caso.} \end{cases}$

Clan (CLIQUE):

Instancia: Una gráfica $G = (V, A)$ y un número $n \leq \text{card}(V)$.

Solución: $\begin{cases} 1 & \text{si } K_n \text{ es una subgráfica de } G, \\ 0 & \text{en otro caso.} \end{cases}$

Conjunto independiente:

Instancia: Una gráfica $G = (V, A)$ y un número $n \leq \text{card}(V)$.

Solución: $\begin{cases} 1 & \text{si } I_n \text{ es una subgráfica de } G, \\ 0 & \text{en otro caso.} \end{cases}$

Los tres problemas anteriores se reducen entre sí debido al evidente

Lema: Para una gráfica $G = (V, A)$ y un subconjunto $U \subset V$ las siguientes tres condiciones son equivalentes a pares:

1. U es un recibrimiento por vértices de G.
2. $V - U$ es un conjunto independiente en G.
3. $V - U$ es un clan en la gráfica complementaria G^c.

Proposición: VC es completo-NP.

Demostración: VC es naturalmente un problema en la clase NP.

Para ver que es difícil-NP reduciremos 3SAT a VC.

Dados

- $\mathbf{X} = \{X_1, \ldots, X_n\}$: conjunto de n variables proposicionales,
- $\mathbf{C} = \{c_1, \ldots, c_m\}$: conjunto de m cláusulas de 3 literales sobre $\mathbf{X}$,

construiremos una instancia de VC de manera que ésta dé una instancia con respuesta positiva en VC si y sólo si la forma conjuntiva $\mathbf{C}$ es satisfactible. En tal caso, un recubrimiento por vértices de la instancia construída ha de determinar una asignación de valores de verdad a las variables en $\mathbf{X}$ que satisfaga a $\mathbf{C}$.

Definamos

- para cada variable $X_j \in \mathbf{X}$, una gráfica consistente de una sola arista,

$$G^j = (\{x_j, \bar{x}_j\}, \{a_j = \{x_j, \bar{x}_j\}\}),$$

- para cada cláusula $c_i \in C$, una gráfica consistente de un triángulo,

$$G_i = (\{u_i, v_i, w_i\}, \{\{u_i, v_i\}, \{v_i, w_i\}, \{w_i, u_i\}\}),$$

y

- para cada cláusula, que consta de tres literales,

$$c_i = \{x_{j_{i1}}^{\epsilon_{j_{i1}}}, x_{j_{i2}}^{\epsilon_{j_{i2}}}, x_{j_{i3}}^{\epsilon_{j_{i3}}}\}$$

tendemos las aristas de los vértices de G_i a los correspondientes vértices "literales" de las gráficas de la forma G^j siguientes

$$\{u_i, x_{j_{i1}}^{\epsilon_{j_{i1}}}\} \quad , \quad \{v_i, x_{j_{i2}}^{\epsilon_{j_{i2}}}\} \quad , \quad \{w_i, x_{j_{i3}}^{\epsilon_{j_{i3}}}\}$$

Así pues, la gráfica construída $G = (V, A)$ consta de $2n + 3m$ vértices y $n + 6m$ aristas.

- Finalmente, hagamos $N = n + 2m$.

Una solución de 3SAT da una solución de VC: Supongamos que $\boldsymbol{\epsilon}$ es una asignación que satisface a C. Construyamos el recubrimiento U siguiente:

- Para cada $j \leq n$,

$$\begin{aligned} \epsilon_j = 0 \quad &\Rightarrow \quad \text{poner } \bar{x}_j \text{ de } G^j \text{ en } U, \\ \epsilon_j = 1 \quad &\Rightarrow \quad \text{poner } x_j \text{ de } G^j \text{ en } U. \end{aligned}$$

- Como la asignación $\boldsymbol{\epsilon}$ satisface a cada una de las cláusulas c_i tenemos que para toda G_i uno de los vértices "queda cubierto", es decir, es el extremo de una arista cuyo otro extremo ha quedado en U. Fijamos pues en cada triángulo G_i un vértice cubierto y los otros dos los incluimos en U.
- Tenemos así un recubrimiento por vértices con K vértices.

Una solución de VC da una solución de 3SAT: Si U es un recubrimiento por vértices con $n+2m$ vértices entonces en cada arista G^j exactamente uno de sus extremos pertenece a U. Hagamos verdadera a la literal correspondiente al extremo en U, en cada G^j. Esta asignación evidentemente satisface a cada cláusula.

8.4 Problemas de asignación de tareas

8.4.1 Problemas con varios procesadores

Nociones básicas

En un proceso de asignación de tareas con varios procesadores, utilizaremos los conceptos y la notación siguientes:

- Un conjunto finito de *tareas*: $A = \{a_1, \ldots, a_n\}$.
- *Duración* (de ejecución) de cada tarea, $\forall a \in A : d(a) \in \mathbf{Z}^+$.
- Un conjunto finito de *procesadores*: $P = \{p_1, \ldots, p_m\}$.
- Un *punto de saturación (deadline)*: $S \in \mathbf{Z}^+$. Usualmente S es una cota para la "carga" que puede recibir cada procesador.
- Una *asignación factible* es una partición $\mathcal{A} = \{A_i\}_{1 \leq i \leq m}$ del conjunto de tareas A tal que

$$\operatorname*{Max}_{1 \leq i \leq m} \left\{ \sum_{a \in A_i} d(a) \right\} \leq S.$$

- Sea "$\preceq$" una relación de orden en A. Una *asignación con restricción en las precedencias* es una función $\sigma : A \to [0, S]$ tal que
  - σ es monótona, i.e.: $a \preceq b \Rightarrow \sigma(a) \leq \sigma(b)$,
  - en ningún momento se excede la capacidad de proceso:

$$\forall i \leq S : \ \operatorname{card}\{a \in A | \sigma(a) \leq i < \sigma(a) + d(a)\} \leq m.$$

Problemas

Asignación a varios procesadores (Multiprocessor scheduling):

Instancia: Un conjunto A finito de tareas, un vector de duraciones de tareas $\mathbf{d} \in N^A$, un número m de procesadores y un punto de saturación S.

Solución: $\begin{cases} 1 & \text{si existe una asignación factible } \mathcal{A} \text{ de } A, \\ 0 & \text{en otro caso.} \end{cases}$

Es completo-NP pues PARTITION se reduce a él. En efecto, una instancia de PARTITION corresponde a una de "Multiprocessor scheduling" con $m = 2$ y $D = \frac{1}{2}\sum_{a \in A} d(a)$.

Otro problema completos-NP es el siguiente:

Asignación con restricción en las precedencias (Precedence constrained scheduling):

Instancia: Un conjunto A finito de tareas, cada una con una duración unitaria, un número m de procesadores, un punto de saturación S y una relación de orden "$\preceq$" en A.

Solución: $\begin{cases} 1 & \text{si hay asignación con restricción en las precedencias,} \\ 0 & \text{en otro caso.} \end{cases}$

8.4.2 Problemas con un solo procesador

Nociones básicas

En un proceso de asignación de tareas con un solo procesador, utilizaremos los conceptos y la notación siguientes:

- Un conjunto finito de *tareas*: $A = \{a_1, \ldots, a_n\}$.
- *Duración* (de ejecución) de cada tarea, $\forall a \in A : d(a) \in \mathbf{Z}^+$.
- Momento de *"lanzamiento"* de cada tarea, $\forall a \in A : r(a) \in \mathbf{Z}^+$.
- Un *instante de saturación (deadline)* en la realización de cada tarea, $\forall a \in A : s(a) \in \mathbf{Z}^+$.
- Una *asignación factible* para A es una función $\sigma : A \to N$ tal que, para toda tarea $a \in A$:
  1. $r(a) \leq \sigma(a)$: el inicio de cada tarea no puede ser anterior a su momento de lanzamiento,
  2. $\sigma(a) + d(a) \leq s(a)$: el inicio más la duración no debe exceder el instante de saturación,
  3. $\forall b \in A : b \neq a \Rightarrow \begin{cases} \sigma(b) + d(b) \leq \sigma(a) \\ \text{ó} \\ \sigma(a) + d(a) \leq \sigma(b) \end{cases}$: Dos tareas distintas no pueden traslapar sus ejecuciones.
- Sea "$\preceq$" una relación de orden en el conjunto de tareas y $K \geq 0$. Una *asignación con retraso acotado* es una asignación

$$\sigma : A \to [0, \text{card}(A) - 1]$$

tal que

1. σ es inyectiva y monótona, i.e.: $a \preceq b \Rightarrow \sigma(a) \leq \sigma(b)$,
2. el conjunto de tareas que "no se ajustan a sus tiempos" no excede de K elementos, i.e.

$$\text{card}\{a \in A | \sigma(a) + d(a) > s(a)\} \leq K.$$

Problemas

Secuenciación en intervalos (Sequencing within intervals):

Instancia: Un conjunto A finito de tareas, un vector de instantes de lanzamiento de tareas $\mathbf{r} \in N^A$, un vector de saturaciones de tareas $\mathbf{s} \in N^A$ y un vector de duraciones de tareas $\mathbf{d} \in N^A$.

Solución: $\begin{cases} 1 & \text{si existe una asignación factible } \sigma : A \to N, \\ 0 & \text{en otro caso.} \end{cases}$

Proposición: "Sequencing within intervals" es completo-NP.

Demostración: Es evidentemente un problema en NP. Para ver que es difícil-NP, reduzcamos PARTITION a él.

Sea $T = \{t_1, \ldots, t_n\}$ una instancia de PARTITION. Se trata de encontrar $J \subset [1, n]$ tal que $\sum_{j \in J} t_j = \sum_{j \in J^c} t_j$. Sea $B = \sum_{j=1}^{n} t_j$.

Construyamos la siguiente instancia de "Sequencing within intervals":

- Por cada sumando t_j, definimos,
  - una tarea a_j, con
  - lanzamiento planeado $r(a_j) = 0$,
  - instante de saturación $s(a_j) = B + 1$, y
  - duración $d(a_j) = t_j$.
- Ponemos una tarea "extra" a_0 con parámetros

$$r(a_0) = \left\lceil \tfrac{B}{2} \right\rceil \quad , \quad s(a_0) = \left\lceil \tfrac{B+1}{2} \right\rceil \quad , \quad d(a_0) = 1$$

  Observamos que

$$s(a_0) - r(a_0) = \begin{cases} 1 & \text{si } B \text{ es par,} \\ 0 & \text{si } B \text{ es impar.} \end{cases}$$

Vemos pues que si B es impar entonces ni PARTITION ni "Sequencing within intervals" tienen solución.

Ahora, si B es par, toda asignación factible σ necesariamente hará $\sigma(a_0) = \frac{B}{2}$, y consecuentemente las tareas restantes se dividen en dos conjuntos:

- las que se realizan antes de a_0, las cuales disponen, en total, de $\frac{B}{2}$ unidades de tiempo para ejecutarse, y
- las que se realizan después de a_0, las cuales disponen, en total, también, de $\frac{B}{2}$ unidades de tiempo para ejecutarse.

Tenemos así una solución de PARTITION.

Recíprocamente, de acuerdo con lo anterior, una solución de PARTITION da una asignación factible.

Mínimos retrasos (Minimum tardiness sequencing):

Instancia: Un conjunto A finito de tareas, cada una con una duración de 1 (es decir, el vector de duraciones es $\mathbf{d} = \mathbf{1} \in N^A$), un vector de saturaciones de tareas $\mathbf{s} \in N^A$, un orden "$\preceq$" en A y un entero $K \leq \text{card}(A)$.

Solución: $\begin{cases} 1 & \text{si existe una asignación con retrasos acotados por } K, \\ 0 & \text{en otro caso.} \end{cases}$

Proposición: "Minimum tardiness sequencing" es completo-NP.

Demostración: Para verlo como un problema difícil-NP reduciremos CLIQUE a él.

Sea pues $[G = (V, E), J \leq \text{card}(V)]$ una instancia de CLIQUE. Construiremos una instancia de "Minimum tardiness sequencing" tal que ésta tendrá una asignación con retrasos acotados por una cota construída si y sólo si la gráfica G contiene a K_J como una subgráfica.

Definamos:

- Tareas: $A = V \cup E$.
- Cota: $K = \text{card}(E) - \frac{1}{2}J(J-1)$.
- Orden de A: $\forall a, b \in A$:
$$a \preceq b \;\Leftrightarrow\; (a \in V) \wedge (b \in E) \wedge (a \in b),$$
es decir, $a \preceq b$ si a es un vértice, b es una arista y a es un vértice extremo de b.

  Con esto tendremos que en toda asignación monótona, cada arista ha de "ejecutarse" después que cualquiera de sus vértices.
- Saturaciones:
$$s : a \mapsto \begin{cases} \frac{1}{2}J(J+1) & \text{si } a \in E, \\ \text{card}(A) & \text{si } a \in V. \end{cases}$$
Así pues las únicas tareas que pueden incurrir en retrasos son las correspondientes a las aristas.

Para una asignación σ, dividamos al conjunto de aristas en dos clases

- las que se realizan a tiempo:
$$\begin{aligned} E_{1\sigma} &= \{a \in E | \sigma(a) + 1 \leq s(a)\} \\ e_{1\sigma} &= \text{card}(E_{1\sigma}) < \frac{1}{2}J(J+1), \end{aligned}$$
- las que no se realizan a tiempo:
$$\begin{aligned} E_{2\sigma} &= \{a \in E | \sigma(a) + 1 > s(a)\} \\ e_{2\sigma} &= \text{card}(E) - e_{1\sigma} \end{aligned}$$

Pues bien, si σ fuese una solución de "Minimum tardiness sequencing" entonces $e_{2\sigma} \leq K$ y consecuentemente $e_{1\sigma} \geq \frac{1}{2}J(J-1)$.

Así pues, el entero $e_{1\sigma}$ debe estar en el intervalo $[\frac{1}{2}J(J-1), \frac{1}{2}J(J+1)[$, el cual contiene precisamente J enteros.

Si $e_{1\sigma} = \frac{1}{2}J(J-1)$ entonces el conjunto de vértices de la gráfica que son extremos de aristas en $E_{1\sigma}$, llamémoslo V_1, por lo menos tiene J vértices. Esta cota mínima se alcanza cuando $E_{1\sigma}$ es una gráfica completa K_J.

Por otro lado, como σ ha de ser monótona, si toda "tarea" de $E_{1\sigma}$ se realiza "antes" de $\frac{1}{2}J(J+1)$ entonces toda "tarea" en V_1 también ha de realizarse antes de ese tiempo.

Así, se tiene que ambos números $e_{1\sigma}$ y $e_{1\sigma} + \text{card}(V_1)$ están en el intervalo $[\frac{1}{2}J(J-1), \frac{1}{2}J(J+1)]$. Esto obliga a que $\text{card}(V_1) = J$ y por tanto los vértices en V_1 forman un clan en G con J elementos.

Recíprocamente, si V_1 es un conjunto de J vértices que forman un clan en G entonces a ellos y a sus aristas les podemos asociar un "tiempo" de ejecución anterior a $\frac{1}{2}J(J+1)$. La asignación así construída provee una solución de "Minimum tardiness sequencing".

8.5 Problemas en teoría de números

8.5.1 Congruencias cuadráticas (CC)

Instancia: Tres enteros positivos $a, b, c \in N$.

Solución: Un entero $x < c$ tal que $x^2 \equiv a \bmod b$.

Referencia:

Manders, Adleman: "NP-complete decision problems for binary quadratics", *J. Comput. System Sci.* **16**, 168-184, 1978.

Comentario: 3SAT se reduce polinomialmente a este problema.

Residuos cuadráticos en general

Para un primo impar p y un entero a primo relativo con p definamos

$$\left(\frac{a}{p}\right) = \begin{cases} +1 & \text{si } \exists x:\ x^2 \equiv a \bmod p, \\ -1 & \text{en otro caso.} \end{cases}$$

Se tiene las propiedades siguientes:

1. $\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \bmod p$
2. $\left(\frac{a}{p}\right)\left(\frac{b}{p}\right) = \left(\frac{ab}{p}\right)$: Operación *multiplicativa*.
3. $\left(\frac{-1}{p}\right) = (-1)^{\frac{p-1}{2}}$
4. $\left(\frac{q}{p}\right)\left(\frac{p}{q}\right) = (-1)^{\frac{p-1}{2}\cdot\frac{q-1}{2}}$: *Ley gaussiana de reciprocidad.*

Ejemplo: Caracterizar a los primos p tales que 3 es residuo cuadrático módulo p.

Se tiene

$$\left(\frac{3}{p}\right) = \left(\frac{p}{3}\right)(-1)^{\frac{p-1}{2}}.$$

Por un lado

$$\left(\frac{p}{3}\right) = \begin{cases} +1 & \text{si } p \equiv 1 \bmod 3, \\ -1 & \text{si } p \equiv 2 \bmod 3. \end{cases}$$

y por otro lado

$$(-1)^{\frac{p-1}{2}} = \begin{cases} +1 & \text{si } p \equiv 1 \bmod 4, \\ -1 & \text{si } p \equiv 3 \bmod 4. \end{cases}$$

Por tanto $\left(\frac{3}{p}\right) = 1$ cuando y sólo cuando

$$[(p \equiv 1 \bmod 3)\&(p \equiv 1 \bmod 4)] \ \text{ o } \ [(p \equiv 2 \bmod 3)\&(p \equiv 3 \bmod 4)],$$

lo cual equivale a que p sea congruente con 1 o con 11 módulo 12.

p $\equiv$ 11 mod 12	x_0	x_1
251	76	175
263	23	240
311	25	286
347	95	252
359	163	196
383	159	224
419	29	390

Tabla 8.1: Raices cuadradas de 3 para primos congruentes con 11 módulo 12.

En la Tabla 1 presentamos las raices cuadradas de 3 para algunos primos congruentes con 11 módulo 12.

Ahora, si a, b son primos relativos, b es impar y

$$b = p_0 \cdot \cdots \cdot p_k$$

es la descomposición en primos de b, definimos

$$\left(\frac{a}{b}\right) = \prod_{i=0}^{k} \left(\frac{a}{p_i}\right).$$

En este caso tenemos que la congruencia

$$x^2 \equiv a \bmod b$$

tiene solución si y sólo si $\forall i$ la congruencia

$$x^2 \equiv a \bmod p_i$$

posee una solución.

Así pues

$$\exists x : \ x^2 \equiv a \bmod b \ \Rightarrow \ \left(\frac{a}{b}\right) = 1,$$

aunque el recíproco no se cumple.

Las propiedades anteriores permiten decidir efectivamente cuándo un número es un residuo cuadrático módulo alguno otro. Sin embargo estas demostraciones ¡NO SON CONSTRUCTIVAS!

Procedimiento de reducción de 3SAT

Sea $\Phi(\mathbf{X}) = \bigwedge\{c | c \in \mathbf{C}\}$ una instancia de 3SAT:

- $\mathbf{X} = \{X_1, \ldots, X_n\}$: variables,
- $\mathbf{C}$: cláusulas con 3 literales

$$\forall c \in \mathbf{C} \, \exists X_i, X_j, X_k \in \mathbf{X}, \epsilon_i, \epsilon_j, \epsilon_k \in \{-1, 1\} : c = X_i^{\epsilon_i} \vee X_j^{\epsilon_j} \vee X_k^{\epsilon_k}.$$

Decimos que X_i, X_j, X_k *aparecen* en c y escribimos $X_i, X_j, X_k \in c$.

Supondremos que Φ no posee cláusulas repetidas ni tautologías.

Enumeremos al conjunto de clásusulas como

$$\mathbf{C}_\Phi = \{c_1, \ldots, c_m\}.$$

Hagamos

$$\mathrm{PC} = -\sum_{i=1}^{m} 8^i = \frac{8}{7}(8^m - 1),$$

y para cada variable X_j definamos

$$\begin{array}{lcll} p_j^+ & := & \sum\{8^i | X_j \in c_i\} & \text{suma de pesos de cláusulas donde } X_j \text{ aparece } \textit{afirmada}. \\ p_j^- & := & \sum\{8^i | \neg X_j \in c_i\} & \text{suma de pesos de cláusulas donde } X_j \text{ aparece } \textit{negada}. \end{array}$$

Sea $q = 2m + n$. Para $k \in [0, q]$ definimos un coeficiente c_k:

$$\begin{array}{rcl} k = 0 & : & \\ & & c_0 = 1 \\ 1 \leq k \leq 2m & : & \\ & & c_k = \left\{ \begin{array}{ll} -\frac{1}{2} 8^{\frac{k+1}{2}} & \text{si } k \text{ es impar,} \\ -8^{\frac{k}{2}} & \text{si } k \text{ es par,} \end{array} \right. \\ 2m + 1 \leq k \leq q & : & \\ & & c_k = \dfrac{1}{2}(p^+_{k-2m} - p^-_{k-2m}) \end{array}$$

Hagamos

$$\mathrm{P} = \mathrm{PC} + \sum_{k=0}^{q} c_k + \sum_{j=1}^{n} p_j^-.$$

Puede verse que existen $1 + q$ coeficientes

$$\alpha_0, \alpha_1, \ldots, \alpha_q \in \{-1, +1\}$$

tales que

$$\mathrm{P} = \sum_{k=0}^{q} c_k \alpha_k.$$

Tomemos los q primos consecutivos $P_0, P_1, \ldots, P_q$ a partir de $P_0 = 13$.

Para cada $k \in [0, q]$ elegimos $t_k \in N$ como el mínimo entero que satisface el sistema de congruencias

$$\begin{array}{rcl} t_k & \equiv & c_k \bmod 8^{m+1} \\ t_k & \equiv & 0 \bmod \displaystyle\prod_{l \in [0,q] \setminus k} P_l^{q+1} \\ t_k & \not\equiv & 0 \bmod P_k \end{array}$$

Por último definimos los coeficientes

$$\begin{array}{rcl} F & = & \displaystyle\sum_{k=0}^{q} t_k \\ D & = & \displaystyle\prod_{k=0}^{q} P_l^{q+1} \\ B & = & 2 \cdot 8^{m+1} \cdot D \\ E & \equiv & (2 \cdot 8^{m+1} + D)^{-1} \bmod B \\ A & \equiv & E \cdot (D\mathrm{P}^2 + 2 \cdot 8^{m+1} \cdot F^2) \bmod B \end{array}$$

Consideramos entonces la instancia del problema (CC) siguiente:

$$T(\Phi): \text{ Encontrar } x \text{ tal que } x^2 \equiv A \bmod B \quad \text{con } 0 \leq x \leq F.$$

Veamos que

$$\Phi \text{ posee una solución } \Leftrightarrow T(\Phi) \text{ posee una solución.}$$

1. Si se tiene n variables $X_1, \ldots, X_n$ entonces se puede tener

$$N\exists C = \binom{n}{3} \cdot 2^3 = \frac{4}{3} n(n-1)(n-2)$$

cláusulas de tres literales.

Para cada tal cláusula

$$c_{i,j,k}^{\epsilon_i,\epsilon_j\epsilon_k} = X_i^{\epsilon_i} \vee X_j^{\epsilon_j} \vee X_k^{\epsilon_k}$$

y cada asignación $A : \{X_1, \ldots, X_n\} \to \{0,1\}$ definimos

$$\begin{aligned} r_A(c_{i,j,k}^{\epsilon_i,\epsilon_j\epsilon_k}) &= A(X_i^{\epsilon_i}) + A(X_j^{\epsilon_j}) + A(X_k^{\epsilon_k}) \\ &: \text{ suma de literales con valor verdadero en } c_{i,j,k}^{\epsilon_i,\epsilon_j\epsilon_k} \end{aligned}$$

Naturalmente

$$A(X_i^{\epsilon_i}) = \begin{cases} A(X_i) & \text{si } \epsilon_i = 1 \\ 1 - A(X_i) & \text{si } \epsilon_i = 0 \end{cases}$$

y

$$r_A(c_{i,j,k}^{\epsilon_i,\epsilon_j\epsilon_k}) \in \{0,1,2,3\}.$$

2. Dada una instancia $\Phi = \bigwedge \mathbf{C}_\Phi$ de 3SAT, se tiene que una asignación $A : \{X_1, \ldots, X_n\} \to \{0,1\}$ *satisface* a Φ si y sólo si para toda cláusula c de tres literales se tiene

$$\begin{aligned} c \in \mathbf{C}_\Phi &\Rightarrow r_A(c) \geq 1 \\ c \notin \mathbf{C}_\Phi &\Rightarrow r_A(c) \geq 0 \end{aligned}$$

equivalentemente

$$\begin{aligned} c \in \mathbf{C}_\Phi &\Rightarrow \exists y_c \in \{0,1,2,3\}(0 = y_c - r_A(c) + 1) \\ c \notin \mathbf{C}_\Phi &\Rightarrow \exists y_c \in \{0,1,2,3\}(0 = y_c - r_A(c)) \end{aligned}$$

Si $\{c_i\}_{i=1,\ldots,m} = \mathbf{C}_\Phi$ es una enumeración de las clásusulas en Φ definimos

$$R_{A,i}(y) = y - r_A(c) + 1.$$

Por lo anterior,

$$A(\Phi) = 1 \Leftrightarrow \forall i \leq m \; \exists y_i \in \{0,1,2,3\}(R_{A,i}(y_i) = 0).$$

3. Como $\forall A \in \{0,1\}^{[1,m]}, i \leq m, y \in [0,3] : -3 \leq R_{A,i}(y) \leq 4$ tenemos

$$\begin{aligned} (\forall i \leq m : \; R_{A,i}(y) = 0) &\Leftrightarrow \sum_{i=1}^{m} R_{A,i}(y) \cdot 8^i = 0 \\ &\Leftrightarrow \sum_{i=1}^{m} R_{A,i}(y) \cdot 8^i \equiv 0 \bmod 8^{m+1}. \end{aligned}$$

4. Ahora bien, la función

$$\begin{aligned} \{-1,+1\}\times\{-1,+1\} &\rightarrow \{0,1,2,3\} \\ (\alpha_1,\alpha_2) &\mapsto \frac{1}{2}(3-\alpha_1-2\alpha_2) \end{aligned}$$

es una biyección.

A cada variable y_i la "parametrizamos" por dos variables $(\alpha_{2i-1},\alpha_{2i})$ haciendo

$$y_i=\frac{1}{2}(3-\alpha_{2i-1}-2\alpha_{2i}).$$

De igual manera, la función

$$\begin{aligned} \{-1,+1\} &\rightarrow \{0,1\} \\ \alpha &\mapsto \frac{1}{2}(1-\alpha) \end{aligned}$$

es una biyección.

A cada valor de verdad $A(X_i)$ lo "parametrizamos" por una variable α_{2m+i} haciendo

$$A(X_i)=\frac{1}{2}(1-\alpha_{2m+i}).$$

$\forall i\leq m$ sea $S_i(\vec{\alpha})$ el resultado de sustituir en $R_{A,i}(y_i)$ las parametrizaciones anteriores.

Entonces tenemos

$$\Phi \text{ es satisfactible } \Leftrightarrow \exists\vec{\alpha}:\sum_{i=1}^{m}S_i(\vec{\alpha})\cdot 8^i\equiv 0 \bmod 8^{m+1}.$$

5. Introduzcamos la ecuación

$$S_i(\vec{\alpha})=\alpha_0+1.$$

Igualmente se tiene

$$\Phi \text{ es satisfactible } \Leftrightarrow \exists\vec{\alpha}:\sum_{i=0}^{m}S_i(\vec{\alpha})\cdot 8^i\equiv 0 \bmod 8^{m+1}.$$

6. De acuerdo con los parámetros introducidos en la construcción de $T(\Phi)$ se tiene

$$\begin{aligned} \sum_{i=0}^{m}S_i(\vec{\alpha})\cdot 8^i\equiv 0 \bmod 8^{m+1} &\Leftrightarrow \sum_{k=0}^{q}c_k\alpha_k\equiv \mathrm{P} \bmod 8^{m+1} \\ &\Leftrightarrow \sum_{k=0}^{q}t_k\alpha_k\equiv \mathrm{P} \bmod 8^{m+1} \end{aligned}$$

7. Ahora, se tiene el siguiente

Lema: Sean

$$F=\sum_{k=0}^{q}t_k \text{ y } D=\prod_{k=0}^{q}P_l^{q+1}$$

como en el algoritmo. La solución del problema

$$\begin{cases} (F+x)(F-x)\equiv 0 \bmod D & \text{con} \\ 0\leq |x|\leq F & x\in Z \end{cases}$$

es de la forma

$$x = \sum_{k=0}^{q} t_k \alpha_k \ , \text{ para algún } \vec{\alpha} \in \{-1,+1\}^{1+q}.$$

8. Así pues, la condición

$$\exists \vec{\alpha} \in \{-1,+1\}^{1+q} : \ \sum_{k=0}^{q} t_k \alpha_k \equiv \mathrm{P} \bmod 8^{m+1}$$

equivale a que exista una solución x del problema

$$Prob_1 : \left\{ \begin{array}{ll} x \equiv \mathrm{P} \bmod 8^{m+1} & \\ (F+x)(F-x) \equiv 0 \bmod D & \text{con} \\ 0 \leq |x| \leq F & x \in Z \end{array} \right.$$

9. Por otro lado, se tiene el siguiente

Lema: Si p es par y $k \geq 3$ entonces

$$(p+x)(p-x) \equiv 0 \bmod 2^{k+1} \ \Leftrightarrow \ \left\{ \begin{array}{c} (p+x) \equiv 0 \bmod 2^k \\ \text{o} \\ (p-x) \equiv 0 \bmod 2^k \end{array} \right.$$

Con esto resulta que el problema $Prob_1$ es equivalente al problema

$$Prob_2 : \left\{ \begin{array}{ll} (\mathrm{P}+x)(\mathrm{P}-x) \equiv 0 \bmod 2 \cdot 8^{m+1} & \\ (F+x)(F-x) \equiv 0 \bmod D & \text{con} \\ 0 \leq |x| \leq F & x \in Z \end{array} \right.$$

10. Conjuntando las dos primeras congruencias del problema $Prob_2$ en una sola, vemos que éste equivale al

$$Prob_3 : \left\{ \begin{array}{rll} \left. \begin{array}{r} \lambda_2 \cdot 2 \cdot 8^{m+1}(F^2 - x_1^2) + \\ + \lambda_3 D(\mathrm{P}^2 - x_1^2) \end{array} \right\} & \equiv 0 \bmod 2 \cdot 8^{m+1} D & \text{con} \\ & 0 \leq x_1 \leq F & x_1 \in Z \text{ y} \\ & \left. \begin{array}{r} \mathrm{mcd}(\lambda_2, D) = 1 \\ \mathrm{mcd}(\lambda_3, 2 \cdot 8^{m+1}) = 1 \end{array} \right\} & x_2, x_3 \in Z \end{array} \right.$$

11. Finalmente, tomamos $\lambda_2 = \lambda_3 = 1$ en $Prob_3$ y obtenemos

$$(2 \cdot 8^{m+1} + D)x^2 \equiv (D\mathrm{P}^2 + 2 \cdot 8^{m+1} F^2) \bmod 2 \cdot 8^{m+1} D,$$

lo cual da propiamente $T(\Phi)$.

12. *Conversión de soluciones*:

[*Solución de* Φ $\Rightarrow$ *Solución de* $T(\Phi)$]: Dada una asignación que satisfaga a Φ calculamos $\vec{\alpha}$ según las ecuaciones en la observación 4. anterior. La solución de $T(\Phi)$ es

$$x = \sum_{k=0}^{q} t_k \alpha_k.$$

[*Solución de* $T(\Phi)$ $\Rightarrow$ *Solución de* Φ]: Dada una solución x de $T(\Phi)$ localizamos $\vec{\alpha}$ tal que

$$x = \sum_{k=0}^{q} t_k \alpha_k,$$

y calculamos la asignación que satisface a Φ según las ecuaciones en la observación 4. anterior.

(CC) es pues completo-NP.

8.5.2 Ecuaciones diofantinas cuadráticas (EDC)

Instancia: Tres enteros positivos $a, b, c \in N$.

Solución: Decidir si acaso existen dos enteros $x_1, x_2 \in N$ soluciones de la ecuación

$$ax_1^2 + bx_2 - c = 0.$$

Referencia:

Manders, Adleman: "NP-complete decision problems for binary quadratics", *J. Comput. System Sci.* **16**, 168-184, 1978.

Comentario: 3SAT se reduce polinomialmente a este problema y el procedimiento de reducción coincide en su mayor parte con la reducción al problema CC anterior.

Ecuaciones diofantinas en general

Recordamos que

- $A \subset N^m$ es *recursivamente enumerable* (r.e.) si existe una función *recursiva* $f : N^{m+n} \to N$ tal que

$$\forall \mathbf{a} \in N^m : \ \mathbf{a} \in A \ \Leftrightarrow \ \exists \mathbf{x} \in N^n : f(\mathbf{a}, \mathbf{x}) = 0.$$

- $A \subset N^m$ es *diofantino* si existe un *polinomio* $p(\mathbf{Z}, \mathbf{X}) \in Z[\mathbf{Z}, \mathbf{X}]$ tal que

$$\forall \mathbf{a} \in N^m : \ \mathbf{a} \in A \ \Leftrightarrow \ \exists \mathbf{x} \in N^n : p(\mathbf{a}, \mathbf{x}) = 0.$$

Se tiene el célebre

Teorema de Matijacewicz: Un conjunto A es r.e. si y sólo si A es diofantino.

Dado que existen conjuntos r.e. que no son recursivos tenemos que no se puede tener procedimiento computable para decidir cuándo una ecuación diofantina posee solución en los enteros, es decir, *el Décimo Problema de Hilbert es irresoluble.*

Refinamientos del Teorema de Matijacewicz muestran que NO se peuede decidir efectivamnete la existencia de soluciones enteras para polinomios de al menos 9 variables.

Entre 3 y 8 variables no se tiene decidida la cuestión de solubilidad efectiva de ecuaciones diofantinas.

Con 2 variables, se tiene para el caso lineal que

$$aX_1 + bX_2 - c = 0 \text{ posee solución entera } \ \Leftrightarrow \ \text{m.c.d.}(a, b) | c,$$

y, para el caso cubico que si $f(X_1, X_2)$ es un polinomio homogéneo de grado 3 entonces la solubilidad en los enteros de la ecuación

$$f(X_1, X_2) = 0$$

es decidible mediante un algoritmo exponencial.

Para grado 2, veremos que el decidir la solubilidad en los enteros de la ecuación

$$aX_1^2 + bX_2 - c = 0$$

es un problema completo-NP.

Procedimiento de reducción de 3SAT

El siguiente procedimiento es similar al anterior.

Sea $\Phi(\mathbf{X}) = \bigwedge\{c|c \in \mathbf{C}\}$ una instancia de 3SAT de m cláusula y n variables. Supondremos que Φ no posee cláusulas repetidas ni tautologías. Enumeremos al conjunto de clásusulas como

$$\mathbf{C}_\Phi = \{c_1, \ldots, c_m\}.$$

Hagamos

$$\mathrm{PC} = -\sum_{i=1}^{m} 8^i = \frac{8}{7}(8^m - 1),$$

y para cada variable X_j

$$p_j^+ := \sum\{8^i | X_j \in c_i\} \quad p_j^- := \sum\{8^i | \neg X_j \in c_i\}$$

Sea $q = 2m + n$. Para cada $k \in [0, q]$ definimos:

$$\begin{aligned} k = 0 \quad &: \quad c_0 = 1 \\ 1 \leq k \leq 2m \quad &: \quad c_k = \begin{cases} -\frac{1}{2}8^{\frac{k+1}{2}} & \text{si } k \text{ es impar,} \\ -8^{\frac{k}{2}} & \text{si } k \text{ es par,} \end{cases} \\ 2m+1 \leq k \leq q \quad &: \quad c_k = \frac{1}{2}(p^+_{k-2m} - p^-_{k-2m}) \end{aligned}$$

Hagamos

$$\mathrm{P} = \mathrm{PC} + \sum_{k=0}^{q} c_k + \sum_{j=1}^{n} p_j^-.$$

Existen $1 + q$ coeficientes

$$\alpha_0, \alpha_1, \ldots, \alpha_q \in \{-1, +1\}$$

tales que

$$\mathrm{P} = \sum_{k=0}^{q} c_k \alpha_k.$$

Tomemos los q primos consecutivos $P_0, P_1, \ldots, P_q$ a partir de $P_0 = 13$.

Para cada $k \in [0, q]$ elegimos $t_k \in N$ como el mínimo entero que satisface el sistema de congruencias

$$\begin{aligned} t_k &\equiv c_k \bmod 8^{m+1} \\ t_k &\equiv 0 \bmod \prod_{l \in [0,q]\setminus k} P_l^{q+1} \\ t_k &\not\equiv 0 \bmod P_k \end{aligned}$$

Definimos los coeficientes

$$\begin{aligned} F &= \sum_{k=0}^{q} t_k \\ D &= \prod_{k=0}^{q} P_l^{q+1} \\ C &= D \cdot \mathrm{P}^2 - (D+1)^3 2 \cdot 8^{m+1} \cdot F^2 \\ B &= -2 \cdot 8^{m+1} \cdot D \\ A &= (D+1)^3 2 \cdot 8^{m+1} + D \end{aligned}$$

Consideramos entonces la instancia del problema (EC) siguiente:

$$T(\Phi): \text{ Encontrar } x_1, x_2 \text{ tales que } Ax_1^2 + Bx_2 - C = 0.$$

Veamos que

$$\Phi \text{ posee una solución } \Leftrightarrow T(\Phi) \text{ posee una solución.}$$

1. Dada una instancia $\Phi = \bigwedge \mathbf{C}_\Phi$ de 3SAT, se tiene que una asignación $A : \{X_1, \ldots, X_n\} \to \{0, 1\}$ *satisface* a Φ si y sólo si, de acuerdo con la Observación 10. de CC el problema $Prob_3$ posee una solución.

2. Tomamos

$$\begin{aligned} \lambda_2 &= (D+1)^3 \\ \lambda_3 &= -1 \end{aligned}$$

en $Prob_3$. Estos valores cumplen con la tercera condición del $Prob_3$. La primera ecuación es equivalente a que para algún x_2 se tenga

$$(D+1)^3 2 \cdot 8^{m+1} \cdot (F^2 - x_1^2) + D(x_1^2 - \mathrm{P}^2) = 2 \cdot 8^{m+1} \cdot F \cdot x_2.$$

En cuanto a la segunda condición tenemos la equivalencia

$$(D+1)^3 2 \cdot 8^{m+1} \cdot (F^2 - x_1^2) + D(x_1^2 - \mathrm{P}^2) \geq 0 \Leftrightarrow 0 \leq x_1 \leq F.$$

Así pues la satisfactibilidad de Φ, equivalente a la solubilidad del $Prob_3$, es equivalente a la solubilidad en N de la ecuación

$$(D+1)^3 2 \cdot 8^{m+1} \cdot (F^2 - x_1^2) + D(x_1^2 - \mathrm{P}^2) - x_2 2 \cdot 8^{m+1} \cdot F = 0,$$

la cual equivale a $T(\Phi)$.

3. *Conversión de soluciones*:

La conversión de soluciones se hace exactamente igual que en CC.

8.5.3 Divisibilidad simultánea de polinomios lineales (DSP)

Instancia: Dos subconjuntos de vectores enteros

$$\begin{aligned} P &= \{\mathbf{a}_i = (\mathbf{a}_{i1}, a_{i,n+1})\}_{i=1,\ldots,m} \subset N^{n+1}, \\ Q &= \{\mathbf{b}_i = (\mathbf{b}_{i1}, b_{i,n+1})\}_{i=1,\ldots,m} \subset N^{n+1}. \end{aligned}$$

Solución: Un vector $\mathbf{x} \in N^n$ tal que

$$\forall i \leq m : \ (\mathbf{a}_{i1} \cdot \mathbf{x}) + a_{i,n+1} | (\mathbf{b}_{i1} \cdot \mathbf{x}) + b_{i,n+1}.$$

Referencia:

Lipshitz: "The Diophantine problem of addition and divisibility", *Trans. Amer. Math. Soc.* **235**, 271-283, 1978.

Comentario: Para cada $m \geq 5$ el problema resulta ser completo-NP y esto se prueba reduciendo a él el problema EDC anterior. Se desconoce actualmente si acaso la unión de estos problemas, es decir el que se obtiene al dejar "abierta" a m, pertenece a NP. El problema es irresoluble en el anillo de enteros algebraicos de extensiones cuadráticas reales.

8.5.4 DSP

Para cada $i \leq m$ consideremos los polinomios lineales

$$\begin{aligned} f_i(\mathbf{X}) &= \mathbf{a}_{i1} \cdot \mathbf{X} + a_{i,n+1} = \sum_{j=1}^{n} a_{ij} X_j + a_{i,n+1} \\ g_i(\mathbf{X}) &= \mathbf{b}_{i1} \cdot \mathbf{X} + b_{i,n+1} = \sum_{j=1}^{n} b_{ij} X_j + b_{i,n+1} \end{aligned}$$

El problema DSP coincide con el de decidir la validez de la fórmula

$$\begin{aligned} &\exists x_1 \ \ldots \ \exists x_n \qquad \bigwedge_{i=1}^{m} [f_i(\mathbf{x}) | g_i(\mathbf{x})] \\ &\exists x_1 \ \ldots \ \exists x_n \qquad \forall i \leq m : [f_i(\mathbf{x}) | g_i(\mathbf{x})] \end{aligned}$$

Para $n = 1$ y $a_{i1} = 0, b_{i1} = 1 \ \forall i \leq m$, el problema queda de la forma

$$\exists x \bigwedge_{i=1}^{m} [a_{i2} | x + b_{i2}].$$

El *Teorema Chino del Residuo* estipula que el anterior problema tiene solución si y sólo si

$$\forall i_1, i_2 \leq m : \ i_1 \neq i_2 \ \Rightarrow \ \text{m.c.d.}(a_{i_1 2}, a_{i_2 2}) | (b_{i_1 2} - b_{i_2 2}).$$

DSP es pues una generalización del Teorema Chino del Residuo. Pero resulta ser completo-NP.

8.5.5 Enteros algebraicos

1. Sea K un campo. Un polinomio $P(X) \in K[X]$ es

   - *irreducible* si rige la implicación

     $$P(X) = Q_1(X) Q_2(X) \ \Rightarrow \ \partial Q_1(X) = 0 \ \vee \ \partial Q_2(X) = 0,$$

   - *mónico* si su coeficiente principal es 1.

   Un elemento α en una extensión de K es *algebraico* sobre K si existe un polinomio $P(X) \in K[X]$ tal que $P(\alpha) = 0$.

   Si α es algebraico existe un polinomio mínimo irreducible $P(X)$ tal que $P(\alpha) = 0$, y es único salvo multiplicación por constantes. $P(X)$, hecho mónico, es *el irreducible de* α.

   α es *entero algebraico* si es algebraico y su polinomio irreducible es mónico con coeficientes enteros.

2. El conjunto de números algebraicos forma un campo.

3. El conjunto de enteros algebraicos forma un anillo.

4. Para todo algebraico α existe un entero racional m tal que $m\alpha$ es un entero algebraico.

5. Si α es algebraico y su irreducible es de orden n el conjunto

   $$K(\alpha) = \left\{ \sum_{i=0}^{n-1} a_i \alpha^i | a_i \in K, \forall i \in [0, n-1] \right\}$$

es una extensión de K de *orden* n.

Las extensiones de orden 2 se dicen ser *cuadráticas*.

6. Dados dos enteros algebraicos α y β decimos que α *divide a* β, $\alpha|\beta$, si existe un entero algebraico γ tal que $\beta = \alpha \cdot \gamma$.

Los divisores de 1 se dicen ser *unidades*.

Las unidades forman un grupo multiplicativo.

7. Si m es un entero racional que no es un cuadrado, entonces $Q(\sqrt{m})$ es un campo cuadrático. El conjunto de enteros algebraicos $Z_{Q(\sqrt{m})}$ se caracteriza de la siguiente forma:

$$\begin{aligned} m \not\equiv 1 \bmod 4 \quad &\Rightarrow \quad Z_{Q(\sqrt{m})} = \left\{a + b\sqrt{m} | a, b \in Z\right\} \\ m \equiv 1 \bmod 4 \quad &\Rightarrow \quad Z_{Q(\sqrt{m})} = \left\{\frac{a + b\sqrt{m}}{2} | a, b \in Z\right\} \end{aligned}$$

8. Un campo *cuadrático real* es de la forma $R(\sqrt{m})$ donde $m \in N$ no es un cuadrado. Sus enteros son las raices en él de polinomios mónicos con coeficientes enteros.

8.5.6 Solubilidad de DSP

Consideremos la versión de DSP consistente en decidir a validez de la fórmula

$$\exists \mathbf{x} \in N^n : \bigwedge_{i=1}^{m} [f_i(\mathbf{x}) | g_i(\mathbf{x})]$$

donde para cada $i \leq m$ $f_i(\mathbf{X}), g_i(\mathbf{X})$ son polinomios lineales con coeficientes enteros (racionales).

1. En esta sección ϕ siempre denotará una fórmula de la forma

$$\phi(\mathbf{x}) \equiv \bigwedge_{i=1}^{m} [f_i(\mathbf{x}) | g_i(\mathbf{x})].$$

$\phi(\mathbf{x})$ se dice ser *positiva* si puede expresarse de manera que todos los coeficientes que aparezcan en los polinomios son números naturales.

2. En lo sucesivo

$$\begin{array}{ll} \mathbf{A} \in Q^{n \times n} & : \text{ es una transformación lineal } N^n \to N^n \\ \mathbf{b} \in Q^n & : \text{ es un vector que determina una } \textit{traslación } N^n \to N^n \end{array}$$

$\mathbf{u} \mapsto \mathbf{x} = \mathbf{A}\ \mathbf{u} + \mathbf{b}$ es una transformación *afín*.

Para una fórmula $\phi(\mathbf{x})$ definimos

$$\psi(\mathbf{u}) \equiv \phi(\mathbf{A}\ \mathbf{u} + \mathbf{b}) \wedge h(\mathbf{u})$$

donde $h(\mathbf{u})$ es una fórmula tal que se tenga

$$\psi(\mathbf{u}) \ \Leftrightarrow \ \phi(\mathbf{A}\ \mathbf{u} + \mathbf{b}) \& [\mathbf{x} = \mathbf{A}\ \mathbf{u} + \mathbf{b} \text{ es un entero algebraico}].$$

Escribiremos $\psi = S_{\mathbf{A},\mathbf{b}}(\phi)$, y diremos que ψ *se obtiene de* ϕ *mediante el cambio de variable* $\mathbf{x} = \mathbf{A}\ \mathbf{u} + \mathbf{b}$.

3. Un *cono* en R^n es un conjunto $C \subset R^n$ tal que

- Contiene al cero: $\mathbf{0} \in C$.
- Es cerrado bajo la suma:
$$\mathbf{x}, \mathbf{y} \in C \ \Rightarrow \ \mathbf{x} + \mathbf{y} \in C.$$
- Contiene al semisegmento positivo de cada uno de sus elementos:
$$\mathbf{x} \in C, t > 0 \ \Rightarrow \ t\mathbf{x} \in C.$$
- No contiene parejas simétricas:
$$\forall \mathbf{x} : \ -\mathbf{x}, \mathbf{x} \in C \ \Rightarrow \ \mathbf{x} = \mathbf{0}.$$

Por ejemplo, el primer "2^n-ante" es un cono.

Si C es un cono la relación "$\leq_C$" definida como

$$\mathbf{x} \leq_C \mathbf{y} \ \Leftrightarrow \ \mathbf{y} - \mathbf{x} \in C,$$

es una relación de orden.

Si C está definido por un predicado

$$C = \{\mathbf{x} | \chi(\mathbf{x})\},$$

diremos que $\chi(\mathbf{x})$ *define un orden.*

4. **Lema:** Sea

$$\omega(\mathbf{x}) \equiv \chi(\mathbf{x}) \wedge (\mathbf{B}\mathbf{x} \leq \mathbf{c}),$$

donde $\chi(\mathbf{x})$ define un orden, $\mathbf{B} \in Q^{m \times n}$ y $\mathbf{c} \in Q^m$.

Entonces existe un conjunto de transformaciones afines

$$\{\mathbf{u} \mapsto \mathbf{x} = \mathbf{A}_i \mathbf{u} + \mathbf{b}_i\}_{1 \leq i \leq I}$$

tal que

$$\exists \mathbf{x} \in N^n(\phi(\mathbf{x}) \wedge \omega(\mathbf{x})) \Leftrightarrow \bigvee_{i=1}^{I} \exists \mathbf{u} \in N^n(S_{\mathbf{A}_i, \mathbf{b}_i}(\phi)(\mathbf{u}))$$

y

$$\forall i \in [1, I] : \ S_{\mathbf{A}_i, \mathbf{b}_i}(\phi)(\mathbf{u}) \text{ es positivo.}$$

5. **Lema:** Dada una fórmula $\phi(\mathbf{x})$ con coeficientes en Z se puede encontrar algorítmicamente un conjunto $\{\chi_i(\mathbf{x})\}_{1 \leq i \leq I}$ de predicados que definen órdenes y un conjunto de transformaciones afines

$$\{\mathbf{u} \mapsto \mathbf{x} = \mathbf{A}_i \mathbf{u} + \mathbf{b}_i\}_{1 \leq i \leq I}$$

tales que

$$\exists \mathbf{x} \in N^n \phi(\mathbf{x}) \Leftrightarrow \bigvee_{i=1}^{I} \exists \mathbf{x} \in N^n \left(S_{\mathbf{A}_i, \mathbf{b}_i}(\phi)(\mathbf{u}) \wedge \chi_i(\mathbf{x}) \right)$$

y

$$\forall i \in [1, I] : \ S_{\mathbf{A}_i, \mathbf{b}_i}(\phi)(\mathbf{u}) \text{ es positivo.}$$

6. Utilizaremos la notación siguiente

$$\begin{aligned}
\phi(\mathbf{x}) &\equiv \bigwedge_{i=1}^{m} [f_i(\mathbf{x}) | g_i(\mathbf{x})] \\
M &= \text{Max}\{|c| | c \text{ es un coeficiente que aparece en } \phi\} \\
\alpha_p &= \lfloor \log_p(Mn) \rfloor + 2 \\
k(\phi, p) &= (p+1)^{4(n+2)\alpha_p m} \\
a | b \bmod p^k &\Leftrightarrow \exists c : ac \equiv b \bmod p^k
\end{aligned}$$

Lema: Si para algún exponente k y algún primo p tenemos que

$$\phi(\mathbf{x}) \text{ posee una solución mod } p^k$$

y

$$\forall i : \ f_i(\mathbf{x}) \not\equiv 0 \bmod p^k$$

entonces

$$\phi(\mathbf{x}) \text{ posee una solución mod } p^{k(\phi,p)}.$$

7. **Lema:** Una fórmula de la forma

$$\phi(\mathbf{u}) \wedge \chi_i(\mathbf{x})$$

donde ϕ es positiva y χ define un orden, tiene una solución en N si y sólo si

$$\begin{aligned} \forall p \quad \in \quad & \text{Primos} \cap \{i | i \leq \text{Max}(m, \text{número de átomos en } \phi)\} : \\ & \phi(\mathbf{x}) \text{ posee una solución mod } p^{k(\phi,p)}. \end{aligned}$$

8. **Teorema:** Existe un algoritmo para decidir la veracidad de una fórmula

$$\exists \mathbf{x} \in N^n : \phi(\mathbf{x}).$$

9. **Corolario:** DSP está en la clase NP.

10. **Proposición:** DSP es completo-NP

Veamos que el problema EDC de Ecuación Diofantina Cuadrática se reduce a DSP.

Dada una instancia

$$\Phi(a, b, c) \equiv \exists x, y \in Z : ax^2 + by - c = 0,$$

tenemos las siguientes relaciones

$$\begin{aligned} \Phi(a, b, c) \quad & \Leftrightarrow \quad \exists x \in Z : b | (-ax^2 + c) \\ & \Rightarrow \quad \exists z \in N : b | (-az + c) \end{aligned}$$

Consideremos entonces la instancia de DSP

$$T(\Phi) \equiv \exists z \in N : b | (-az + c).$$

Si $T(\Phi)$ se resuelve, se prueba cada una de las posibles soluciones z, las cuáles forman un conjunto finito en la región determinada por el Teorema en la observación anterior y se ve si alguna es un cuadrado, en cuyo caso Φ posee efectivamente una solución.

8.5.7 Algunos otros problemas

Incongruencias simultáneas

Instancia: Un subconjunto de parejas $P = \{(a_i, b_i)\}_{i=1,\ldots,n} \subset N^2$.

Solución: Un entero x tal que

$$\forall i \leq n : \ x \not\equiv a_i \bmod b_i.$$

Referencia:

Stockmeyer, Meyer: "Word problems requiring exponential time", *Proc. 5th Ann. ACM Symp. on Theory of Computing*, ACM New York, 1-9, 1973.

Comentario: 3SAT se transforma en este problema.

Comparación de factores de divisibilidad

Instancia: Dos vectores $\mathbf{a} = (a_1, \ldots, a_n) \in N^n$ y $\mathbf{b} = (b_1, \ldots, b_m) \in N^m$.

Solución: Un entero c tal que

$$\operatorname*{Min}_{c|a_i} i \geq \operatorname*{Min}_{c|b_i} i.$$

Referencia:

Plaisted: "Some polynomial and integer divisibility problems are NP-hard", *Proc. 17th Ann. Symp. on Foundations of Computer Science* IEEE Computer Society, 264-267, 1976.

Comentario: 3SAT se transforma en este problema. Se mantiene en la clase NP aún cuando los números a_i o b_i son distintos a pares.

Divisibilidad de expresiones exponenciales

Instancia: Dos vectores $\mathbf{a} = (a_1, \ldots, a_n) \in N^n$ y $\mathbf{b} = (b_1, \ldots, b_m) \in N^m$ y un entero $q \in N$.

Solución: Decidir si acaso

$$\prod_{i=1}^{n}(q^{a_i} - 1) | \prod_{j=1}^{m}(q^{b_j} - 1).$$

Referencia:

Plaisted: "Some polynomial and integer divisibility problems are NP-hard", *Proc. 17th Ann. Symp. on Foundations of Computer Science* IEEE Computer Society, 264-267, 1976.

Comentario: 3SAT se transforma en este problema. Se desconoce si acaso este problema está en NP o en co-NP. Sin embargo es *seudo-polinomial en tiempo*, es decir, de complejidad polinomial si sus entradas se escriben en unario. Aún para cada q, con $|q| > 1$, y suponiendo que los coeficientes a_i o b_i son productos de primos distintos, se tiene un problema difícil-NP.

Ecuaciones algebraicas en el campo de Galois GF[2]

Instancia: m polinomios $P_i(X_1, \ldots, X_n) \in \mathrm{GF}[2][\mathbf{X}], i = 1, \ldots, m$.

Solución: Decidir si acaso existe $\mathbf{x} \in \mathrm{GF}[2]^n$ que sea una raiz común de los polinomios dados, es decir, tal que

$$P_i(\mathbf{x}) = 0, \ \forall i = 1, \ldots, m.$$

Referencia:

Fraenkel, Yesha: "Complexity of problems in games, graphs and algebraic equations", manuscrito inédito, 1977.

Comentario: El problema X3C[1] se transforma a este problema. Se mantiene completo-NP aún cuando se exige que cada término en los polinomios involucre a lo sumo dos variables. Sin embargo, si todos los polinomios son lineales, o si tan sólo se tiene un polinomio, el problema es polinomial.

Los restantes problemas en esta sección tienen como fuente a la siguiente

Referencia:

Plaisted: "Sparce complex polynomials and polynomial reducibility", *J. Comput. System Sci.* **14**, 210-221, 1977.

No-divisibilidad de un producto de polinomios

Instancia: m sucesiones de parejas $A_i = \{(a_{ij}, b_{ij})\}_{j=1,\ldots,n} \subset Z^2$, con $b_{ij} \geq 0$, y un entero $n_0 \in N$.

Solución: Decidir si acaso

$$(X^{n_0} - 1) | \prod_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} X^{b_{ij}} \right).$$

Comentario: 3SAT se transforma en este problema. Aquí, el demostrar la pertenencia a NP es la parte más complicada.

El siguiente problema similar es también completo-NP:

Instancia: m parejas $A = \{(a_i, b_i)\}_{i=1,\ldots,m} \subset Z^2$ tales que $b_{ij} \geq 0$.

Solución: Decidir si acaso

$$\prod_{i=1}^{m} (X^{a_i} - 1) | \prod_{i=1}^{m} (X^{b_i} - 1).$$

MCD no trivial

Instancia: m sucesiones de parejas $A_i = \{(a_{ij}, b_{ij})\}_{j=1,\ldots,n} \subset Z^2$ tales que $b_{ij} \geq 0$.

Solución: Decidir si acaso

$$\text{MCD} \left\{ \sum_{j=1}^{n} a_{ij} X^{b_{ij}} | i = 1, \ldots, m \right\} \text{ posee grado positivo.}$$

---

[1] **Recubrimiento exacto por tripletas**
Dado un conjunto A una *tripleta* es un subconjunto de A con exactamente 3 elementos.

Instancia: Un conjunto A, cuya cardinalidad es un múltiplo de 3, y una colección $\mathcal{C}$ de tripletas en A.
Solución: Decidir si acaso existe un subconjunto de $\mathcal{C}$ que sea una partición de A.

Comentario: 3SAT se transforma en este problema. Se ignora si acaso pertenece a NP o a co-NP. Se mantiene como un problema difícil-NP aún cuando $\forall i, j : a_{ij} \in \{-1, +1\}$, o $m = 2$. El problema es, sin embargo, seudo-polinomial en tiempo. En la misma situación está el siguiente problema:

Instancia: m sucesiones de parejas $A_i = \{(a_{ij}, b_{ij})\}_{j=1,\ldots,n} \subset Z^2$ tales que $b_{ij} \geq 0$, y $K > 0$.

Solución: Decidir si acaso

$$\text{MCD}\left\{\sum_{j=1}^{n} a_{ij} X^{b_{ij}} | i = 1, \ldots, m\right\} \text{ posee grado a lo sumo } K.$$

Raices de módulo 1

Instancia: Una sucesión de parejas $A = \{(a_i, b_i)\}_{i=1,\ldots,m} \subset Z^2$, con $b_{ij} \geq 0$.

Solución: Decidir si acaso

$$\sum_{i=1}^{m} a_i X^{b_i}$$

posee una raiz en la circunferencia unitaria del plano complejo.

Comentario: 3SAT se transforma a este problema. Se ignora si acaso está en NP o en co-NP.

Número de raices para un producto de polinomios

Instancia: m sucesiones de parejas $A_i = \{(a_{ij}, b_{ij})\}_{j=1,\ldots,n} \subset Z^2$ tales que $b_{ij} \geq 0$ y un entero positivo $n_0 \in N$.

Solución: Decidir si acaso

$$\prod_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij} X^{b_{ij}}\right)$$

tiene a lo sumo n_0 raices complejas.

Comentario: 3SAT se transforma en este problema. Se ignora si acaso pertenece a NP o a co-NP.

Soluciones periódicas de relaciones recurrentes

Instancia: Una sucesión $(c_i, b_i)_{i=1,\ldots,m} \subset Z \times N^+$.

Solución: Decidir si acaso existe una sucesión de $n \geq \text{Max}_{i \leq m}\, b_i$ enteros

$$a_0, a_1, \ldots, a_{n-1}$$

tal que al definir $\forall i \geq n$

$$a_i = \sum_{j=1}^{m} c_j a_{(i-b_j)}$$

se tiene que los a_i's se repiten cada n índices:

$$\forall i, j \geq n : \; i \equiv j \bmod n \;\Rightarrow\; a_i = a_j.$$

Comentario: 3SAT se transforma en este problema. Se ignora si acaso pertenece a NP o a co-NP.

Capítulo 9

Complejidad en redes de Petri generalizadas

9.1 Nociones básicas

Una *red de Petri generalizada*, (r.P.g.), es un sistema $R = (L, T, A, p, M)$ tal que

- $(\{L, T\}, A, p)$ es una gráfica dirigida, bipartita y ponderada, i. e.:
  - $L \cup T$ es el conjunto de vértices,
    - L es un conjunto de *lugares*, digamos $L = \{l_1, \ldots, l_n\}$,
    - T es un conjunto de *transiciones*, digamos $T = \{t_1, \ldots, t_m\}$,
  - A es el conjunto de *aristas*,
    $$A \subset (L \times T) \cup (T \times L),$$
  - $p : A \to N^+$ es una función de *pesos* o de *incidencia*. Un valor nulo de p en una arista es equivalente a la inexistencia de esa arista.
- $M : L \to N$ es una función de *marcado*.

Utilizaremos la terminología siguiente:

- $\mathbf{m} = [M(l_1), \ldots, M(l_n)]$ es el *vector de marcado* de la r.P.g.
- Para cada transición $t \in T$ definimos
  - Nociones de *entrada de* t:
    $$\begin{array}{rcll} \mathit{Ent}(t) & = & \{l \in L | (l,t) \in A\} & : \text{lugares de entrada,} \\ \mathbf{e}_t & = & [p(l_1,t), \ldots, p(l_n,t)] & : \text{vector de entrada.} \end{array}$$
  - Nociones de *salida de* t:
    $$\begin{array}{rcll} \mathit{Sal}(t) & = & \{l \in L | (t,l) \in A\} & : \text{lugares de salida,} \\ \mathbf{s}_t & = & [p(t,l_1), \ldots, p(t,l_n)] & : \text{vector de salida.} \end{array}$$
- Una transición $t \in T$ es *disparable* si
  $$\forall l \in \mathit{Ent}(t) : \ M(l) \geq p(l,t),$$
  o, equivalentemente, si $\mathbf{m} \geq \mathbf{e}_t$.

- Si t es disparable, al dispararse modifica las marcas de sus lugares aledaños:

$$\begin{aligned} l \in Ent(t) &\Rightarrow M'(l) = M(l) - p(l,t) \\ l \in Sal(t) &\Rightarrow M'(l) = M(l) + p(t,l) \end{aligned}$$

  en otras palabras, cuando t se dispara origina la modificación de marcado en la r.P.g.

$$\Phi_t(\mathbf{m}) = \mathbf{m}' = \mathbf{m} - \mathbf{e}_t + \mathbf{s}_t.$$

- Sea $M_0 = \mathbf{m}_0$ un marcado inicial. Una *sucesión de disparo* es una sucesión de transiciones $\mathbf{t} = t_{j_1} \cdots t_{j_k} \in T^*$ tal que
  - t_{j_1} es disparable con el marcado inicial $M_0 = \mathbf{m}_0$ y origina un marcado $\mathbf{m}_1 = \Phi_{t_{j_1}}(\mathbf{m}_0)$.
  - Para cada $\kappa < k$, la transición $t_{j_{\kappa+1}}$ es disparable con el marcado $\mathbf{m}_\kappa$ y ha de originar al dispararse al marcado $\mathbf{m}_{\kappa+1} = \Phi_{t_{j_{\kappa+1}}}(\mathbf{m}_\kappa)$

  En este caso el marcado $\mathbf{m}_k$ se dice ser el *derivado* po $\mathbf{t}$ a partir de $\mathbf{m}_0$. Escribamos

$$\Phi_{\mathbf{t}}(\mathbf{m}_0) = \mathbf{m}_k.$$

- Denotamos al conjunto de sucesiones de disparo como

$$S_R(\mathbf{m}_0) = \{\mathbf{t} | \mathbf{t} \text{ es una sucesión de disparo a partir de } \mathbf{m}_0\}.$$

- El conjunto de vectores de marcado que se obtienen por sucesiones de disparo es el *conjunto de accesibilidad*,

$$Ac_R(\mathbf{m}_0) = \{\Phi_{\mathbf{t}}(\mathbf{m}_0) | \mathbf{t} \in S_R(\mathbf{m}_0)\}.$$

- Dado $\mathbf{m} \in Ac_R(\mathbf{m}_0)$ el conjunto de sucesiones de disparo que *conducen de* $\mathbf{m}_0$ *a* $\mathbf{m}$ es

$$T_R(\mathbf{m}_0, \mathbf{m}) = \{\mathbf{t} \in S_R(\mathbf{m}_0) | \Phi_{\mathbf{t}}(\mathbf{m}_0) = \mathbf{m}\}.$$

- Sea $\mathbf{t} \in T^*$ una sucesión de transiciones. La *vara* de $\mathbf{t}$ es

$$V(\mathbf{t}) = \text{Min}\{\mathbf{m}_0 | \mathbf{t} \in S_R(\mathbf{m}_0).$$

- Si $\mathbf{t} \in S_R(\mathbf{m}_0)$ y $\mathbf{m} = \Phi_{\mathbf{t}}(\mathbf{m}_0)$, el *cambio de marcado* es

$$\Delta(\mathbf{t}) = \mathbf{m} - \mathbf{m}_0.$$

Denotemos por *nil* a la palabra vaciía, y por $\mathbf{0} \in R^n$ al vector cero.

Es fácil ver que la vara de toda sucesión de disparo se ajusta a las recurrencias

$$\begin{aligned} V(nil) &= \mathbf{0} \\ \forall \mathbf{t} \in T^*, t \in T: \quad V(\mathbf{t} \cdot t) &= \text{Max}\{V(\mathbf{t}), \mathbf{e}_t - \Delta(\mathbf{t})\} \end{aligned}$$

y, correspondientemente, el cambio de marcado satisface

$$\begin{aligned} \Delta(nil) &= \mathbf{0} \\ \forall \mathbf{t} \in T^*, t \in T: \quad \Delta(\mathbf{t} \cdot t) &= \Delta(\mathbf{t}) - \mathbf{e}_t + \mathbf{s}_t \end{aligned}$$

Una r.P.g. R se dice ser

ordinaria si su función de pesos asigna, en cada arista, el valor 1 si existe la arista y 0 en otro caso,

sin ciclos unitarios si para todos $l \in L$ y $t \in T$ rigen las implicaciones

$$\begin{aligned}(l,t) \in A &\Rightarrow (t,l) \notin A\\ (t,l) \in A &\Rightarrow (l,t) \notin A\end{aligned}$$

restringida si es ordinaria sin ciclos unitarios.

Para una r.P.g. R definiremos los conceptos siguientes:

marcados accesibles: $\mathbf{m}$ es *accesible* desde el marcado inicial $\mathbf{m}_0$ si $\mathbf{m} \in Ac_R(\mathbf{m}_0)$.

marcados recubribles: $\mathbf{m}$ es *recubrible* desde el marcado inicial $\mathbf{m}_0$ si

$$\exists \mathbf{m}_1 \in Ac_R(\mathbf{m}_0) : \ \mathbf{m} \leq \mathbf{m}_1.$$

lugares acotados: $l = l_i \in L$ es un lugar *acotado* si

$$\exists C > 0 [\forall \mathbf{m} \in Ac_R(\mathbf{m}_0) : |m_i| \leq C].$$

R es *acotada* para un marcado inicial $\mathbf{m}_0$ si todos sus lugares se mantienen acotados desde ese marcado.

transiciones potencialmente disparables: $t \in T$ es una transición *potencialmente disparable* con el marcado $\mathbf{m}$ si existe $\mathbf{t} \in T_R(\mathbf{m}_0, \mathbf{m})$ que contiene a t.

marcados muertos-t: para una transición t el marcado $\mathbf{m}$ está *muerto-t* si t no es potencialmente disparable con el marcado $\mathbf{m}$,

transiciones vivas: $t \in T$ es una transición *viva* en la r.P.g. R, para el marcado inicial $\mathbf{m}_0$, si es potencialmente disparable con todo marcado que sea accesible.

La red misma R está *viva* con el marcado inicial $\mathbf{m}_0$ si toda transición está viva para ese marcado inicial.

transiciones persistentes: $t \in T$ es una transición *persistente* en la r.P.g. R, para el marcado inicial $\mathbf{m}_0$, si la única manera de inhabilitarla es mediante su propio disparo. En otras palabras, si se cumple que $\forall t_1 \in T - \{t\}, \mathbf{m} \in Ac_R(\mathbf{m}_0)$:

$$(\mathbf{m} \geq \mathbf{e}_t) \wedge (\mathbf{m} \geq \mathbf{e}_{t_1}) \ \Rightarrow \ \mathbf{m} \geq \mathbf{e}_t + \mathbf{e}_{t_1}.$$

La red misma R es *persistente* con el marcado inicial $\mathbf{m}_0$ si toda transición es persistente para ese marcado inicial.

9.2 Ejemplo

Consideremos la r.P.g. R descrita como sigue:

- *Lugares*: $L = \{l_1, l_2, l_3\}$.
- *Transiciones*: $T = \{t_1, t_2, t_3, t_4\}$.
- *Pesos de las aristas en* $L \times T$:

$$p_{L\times T} = \begin{bmatrix} 1 & 3 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 1 & 5 \end{bmatrix}$$

Así pues, los vectores de entrada son las columnas de la matriz anterior.

	Posee propiedad	No posee
Accesible desde $\mathbf{m}_0$	[0 99 3]	[0 98 2]
Recubrible desde $\mathbf{m}_0$	[0 98 2]	[0 0 5]
Acotado con $\mathbf{m}_0$	l_1	l_2
Disparable con $\mathbf{m}_0$	t_2	t_3
Potencialmente disparable con $\mathbf{m}_0$	t_3	t_4
Vivo con $\mathbf{m}_0$	t_1	t_3
Persistente con $\mathbf{m}_0$	t_3	t_2
Muerto-t_4	[5 1 0]	[8 0 0]

Figura 9.1: Resumen de propiedades

- *Pesos de las aristas en* $T \times L$:

$$p_{T\times L} = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

  Así pues, los vectores de salida son los renglones de la matriz anterior.

- *Marcado inicial*: $\mathbf{m}_0 = [\ 5 \quad 3 \quad 0\ ]$.

Observamos que la red posee ciclos unitarios, no es ordinaria y, consecuentemente, no es restringida.

En la Figura 9.1 se ejemplifica algunos otros conceptos.

9.3 Problemas de acotación

Sea R una r.P.g. y sea $\mathbf{m}_0$ un marcado inicial. Un *árbol de recubrimiento* $\mathbf{A}(R, \mathbf{m}_0)$ describe las "computaciones" derivables por la red a partir del marcado inicial. Formalmente este árbol se define de manera inductiva:

1. La *raíz* se etiqueta con el marcado inicial $\mathbf{m}_0$.
2. Si $\mathbf{n}$ es un nodo ya construído con etiqueta $\mathbf{m_n}$ pueden ocurrir cuatro casos:
   (a) Ninguna transición es disparable con $\mathbf{m_n}$: En este caso, $\mathbf{n}$ es una *hoja de terminación muerta.*
   (b) El marcado actual se ha repetido previamente: Si existe un ancestro $\mathbf{n}_1$ de $\mathbf{n}$ con la misma etiqueta, es decir, tal que $\mathbf{m}_{\mathbf{n}_1} = \mathbf{m_n}$, entonces $\mathbf{n}$ es una *hoja de terminación ciclada-*λ, y se coloca un *apuntador-*λ de $\mathbf{n}$ a $\mathbf{n}_1$.
   (c) El marcado actual cubre propiamente a uno aparecido previamente: Si existe un ancestro $\mathbf{n}_1$ de $\mathbf{n}$ tal que $\mathbf{m}_{\mathbf{n}_1} < \mathbf{m_n}$, entonces $\mathbf{n}$ es una *hoja de terminación ciclada-*ω, y se coloca un *apuntador-*ω de $\mathbf{n}$ a $\mathbf{n}_1$.
   (d) En cualquier otro caso, para cada transición t disparable se le asocia a $\mathbf{n}$ un nodo hijo etiquetado por el marcado $\mathbf{m_n} - \mathbf{e}_t + \mathbf{s}_t$. El arco hacia el hijo se etiqueta con la transición t.

Las siguientes observaciones son inmediatas

1. Dados R y $\mathbf{m}_0$, $\mathbf{A}(R, \mathbf{m}_0)$ es efectivamente constructible.
2. Cada nodo en $\mathbf{A}(R, \mathbf{m}_0)$ tiene a lo sumo un número finito de hijos. Luego, si el árbol fuera infinito, necesariamente ha de tener ramas infinitas.
3. Por lo anterior, se tiene que todo árbol $\mathbf{A}(R, \mathbf{m}_0)$ es finito.
4. Si $\mathbf{m}$ aparece como etiqueta de un nodo entonces cae en $Ac_R(\mathbf{m}_0)$.

Proposición: Se puede decidir efectivamente si acaso la red R es acotada.

Demostración: La red R es acotada si y sólo si $\mathbf{A}(R, \mathbf{m}_0)$ no posee hojas de terminación ciclada-ω.

Consecuencias: Con técnicas similares a la anterior resulta que los conceptos siguientes son todos efectivamente decidibles:

1. Disparabilidad potencial de transiciones.
2. Para cada transición t, la mortalidad-t de marcados.
3. Disparabilidad infinita de una transición.
4. Para cada lugar l, el acceso a marcados que lo hagan asumir marcas positivas (llegadas de *estafetas*).

9.4 Problemas de acceso

Para una r.P.g. $R = (L, T, A, p)$ y un marcado inicial $\mathbf{m}_0$, consideremos los problemas siguientes:

Problema de acceso (PA):

Instancia: Un vector de marcado $\mathbf{m} \in (N \cup \{\omega\})^L$.

Solución: $\begin{cases} 1 & \text{si } \mathbf{m} \in Ac_R(\mathbf{m}_0), \\ 0 & \text{en otro caso.} \end{cases}$

Problema de acceso a submarcados (PAS):

Instancia: Un subconjunto $L_1 \subset L$ de lugares y un vector de (sub)-marcado $\mathbf{m}_1 \in R^{L_1}$ restringido a L_1.

Solución: $\begin{cases} 1 & \text{si } \exists \mathbf{m} \in Ac_R(\mathbf{m}_0) : \ \mathbf{m}|_{L_1} = \mathbf{m}_1, \\ 0 & \text{en otro caso.} \end{cases}$

Problema de acceso a cero (PA0): PA con la instancia $\mathbf{0}$.

Problema de acceso a cero en un lugar dado (PA0L):

Instancia: Un lugar $l \in L$.

Solución: $\begin{cases} 1 & \text{si } \exists \mathbf{m} \in Ac_R(\mathbf{m}_0) : \ [\mathbf{m}]_l = 0, \\ 0 & \text{en otro caso.} \end{cases}$

Se ve directamente que valen las reducibilidades siguientes:

$$\begin{array}{ccc} \mathbf{PA0} & \overset{instancia}{\hookrightarrow} & \mathbf{PA} \\ \vdots & \ddots & \downarrow instancia \\ \mathbf{PA0L} & \overset{instancia}{\hookrightarrow} & \mathbf{PAS} \end{array}$$

Para completar el diagrama de reducibilidades veamos lo siguiente:

PAS se reduce a PA0: Dada una instancia de **PAS** consistente de una red $R = (L, T, A, p)$, un marcado inicial $\mathbf{m}_0$, un subconjunto $L_1 \subset L$ de lugares y un vector de (sub)-marcado $\mathbf{m}_1 \in R^{L_1}$ la transformaremos en una instancia equivalente de **PA0**.

Introducimos como elementos nuevos

l_0	:	un nuevo lugar tal que -se conecta con todas las transiciones de R mediante ciclos unitarios, y -se le asocia la marca 1,
$\forall l \in L : t_l$	:	una transición propia de l con la arista de peso 1, (l, t_l),
λ_0	:	un nuevo lugar con las aristas de peso 1, $(\lambda_0, t_l), (t_l, \lambda_0),\ \forall l$,
t_0	:	una nueva transición tal que -(l_0, t_0) y (t_0, λ_0) son aristas de peso 1.
t'_0	:	una nueva transición tal que -(λ_0, t'_0) es una arista de peso 1.
$\forall l \in L_1 : \lambda_l$	:	un nuevo lugar, propio de l, con marca $[\mathbf{m}_1]_l$ y con la arista de peso 1, (λ_l, t_l).

Se ve que la instancia transformada tiene una solución positiva en **PA0** si y sólo si la instancia dada tiene también una solución positiva en **PAS**.

PA0 se reduce a PA0L: Dada una instancia de **PA0** consistente de una red $R = (L, T, A, p)$ y un marcado inicial $\mathbf{m}_0$, añadimos un lugar nuevo λ_0 tal que en todo marcado accesible $\mathbf{m}'$ se tenga

$$[\mathbf{m}']_{\lambda_0} = \sum_{l \neq \lambda_0} [\mathbf{m}']_l \,.$$

Así tendremos que la instancia dada da respuesta positiva con **PA0** si y sólo si la instancia modificada la da con **PA0L** para λ_0.

Para esto, inicialmente se le asigna a λ_0 una marca de $\sum_{l \in L} [\mathbf{m}_0]_l$.

Luego, para cada transición $t \in T$ calculamos el cambio de marcas cuando se dispara t:

$$\Delta_t = \sum_{l \in L} \left([\mathbf{s}_t]_l - [\mathbf{e}_t]_l \right).$$

Dependiendo de este valor se hacen conexiones a λ_0:

$$\begin{aligned} \Delta_t \geq 0 \quad &\Rightarrow \quad \text{se incorpora la arista } (t, \lambda_0) \text{ con peso } \Delta_t, \\ \Delta_t < 0 \quad &\Rightarrow \quad \text{se incorpora la arista } (\lambda_0, t) \text{ con peso } -\Delta_t, \end{aligned}$$

Así pues, los cuatro problemas anteriores de acceso son reducibles unos a otros.

9.4.1 Problema de acceso generalizado

Sea $A \subset R^L$. Consideremos el siguiente

Problema de acceso a A ($\mathbf{PA}_A$):

Instancia: Una r.P.g. $R = (L, T, A, p)$ y un marcado inicial $\mathbf{m}_0$

Solución: $\begin{cases} 1 & \text{si } A \cap Ac_R(\mathbf{m}_0) \neq \emptyset, \\ 0 & \text{en otro caso.} \end{cases}$

A es *resoluble por acceso* si $\mathbf{PA}_A$ se reduce a $\mathbf{PA}$.

Se tiene,

1. Todo conjunto accesible es por sí mismo resoluble por acceso.
2. *Problema del marcado común*: La intersección de dos conjuntos resolubles por acceso es también resoluble por acceso.
3. La unión finita de conjuntos resolubles por acceso es también resoluble por acceso.
4. Todo conjunto *lineal*, es decir, toda imagen de una transformación afín,

$$\{\mathbf{m}_0 + \mathbf{A}\mathbf{u} | \mathbf{u} \in R^m\},\ \mathbf{m}_0 \in R^L, \mathbf{A} \in R^{L \times m},$$

   es resoluble por acceso.
5. Todo conjunto *semilineal*, es decir, expresable como la unión finita de conjuntos lineales, es resoluble por acceso.
6. Soluciones de ecuaciones diofantinas lineales y poliedros son resolubles por acceso.

Capítulo 10

Algoritmos probabilísticos y su jerarquía

10.1 Algoritmos probabilísticos

Los algoritmos probabilísticos proceden en algunos momentos según los valores que tome una variable aleatoria.

10.1.1 Expresiones polinomiales nulas

Sea $\mathbf{X} = \{X_1, \ldots, X_m\}$ un conjunto de m variables. Definimos la clase de *expresiones polinomiales enteras* $EPE(\mathbf{X})$, y el *grado* de cada uno de sus elementos, de manera inductiva como sigue:

$$
\begin{array}{lll}
1. & c \in Z & \Rightarrow \quad c \in EPE(\mathbf{X}), \quad \partial(c) = 0 \\
2. & X \in \mathbf{X} & \Rightarrow \quad X \in EPE(\mathbf{X}), \quad \partial(X) = 1 \\
3. & F, G \in EPE(\mathbf{X}) & \Rightarrow \quad F \pm G \in EPE(\mathbf{X}), \\
 & & \qquad \partial(F \pm G) = \mathrm{Max}\{\partial(F), \partial(G)\} \\
4. & F, G \in EPE(\mathbf{X}) & \Rightarrow \quad FG \in EPE(\mathbf{X}), \\
 & & \qquad \partial(FG) = \partial(F) + \partial(G)
\end{array}
$$

Sea

$$\mathrm{Cero} = \{E \in EPE(\mathbf{X}) | E \equiv 0\}.$$

Consideremos el problema siguiente:

Instancia: $E \in EPE(\mathbf{X})$

Solución: $\begin{cases} 1 & \text{si } E \in \mathrm{Cero} \\ 0 & \text{si } E \notin \mathrm{Cero} \end{cases}$

Observaciones: 1. Desarrollando el polinomio dado como una suma de monomios, ordenados lexicográficamente, se resuelve el problema en tiempo exponencial.

2. Si $F(X_1, \ldots, X_m) \notin \mathrm{Cero}$ y $N \geq c \cdot \partial(F)$ entonces F posee a lo sumo $c^{-1}N^m$ raices enteras en el rectángulo $[1, N]^m$.

El problema se resuelve probabilísticamente por el algoritmo siguiente:

$N := 2 \cdot \partial E$;
Choose m integer vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\} \subset [1, N]^m$;
if $\bigwedge_{i=1}^{m}(E(\mathbf{x}_i) = 0)$ **then** "Yes" **else** "No"

Si la entrada E es efectivamente nula el algoritmo responde correctamente.

Si E no es nula el algoritmo responde equivocadamente, con una probabilidad pequeña, la de elegir precisamente m raices de E en un conjunto con $2^m \cdot (\partial E)^m$ puntos, la cual probabilidad es, en todo caso, menor que $\frac{1}{2}$.

La probabilidad de error puede disminuirse arbitrariamente reiterando la selección de puntos de prueba:

$N := 2 \cdot \partial E$;
Flag := *True* ; ntimes := 1 ;
whileFlag $\wedge$ (ntimes $\leq t$) **do** {
  Choose m integer vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\} \subset [1, N]^m$;
  Flag := Flag $\wedge \bigwedge_{i=1}^{m}(E(\mathbf{x}_i) = 0)$;
  ntimes := ntimes + 1 } ;
if Flag **then** "Yes" **else** "No"

En este caso la probabilidad de error es $\frac{1}{2^t}$.

10.1.2 Método de Monte-Carlo para probar primicidad

El siguiente algoritmo, debido a Solovay y Strassen, decide probabilísticamente si acaso n es un número primo.

Input: An odd integer
$n \in N$.

Output: $\begin{cases} \text{Yes} & \text{if } n \text{ is (presumably) prime} \\ \text{No} & \text{Otherwise} \end{cases}$

Even more

$$\text{Prob}(\text{Yes}|n \notin \text{Primes}) < \frac{1}{2}.$$

Choose $a \in [1, n-1]$;
$x := \gcd(a, n)$;
if $x > 1$ **then** "No"
else {
  $\epsilon := a^{\frac{(n-1)}{2}} \bmod n$;
  $\delta := \left(\frac{a}{n}\right)$;
  if $\epsilon = \delta$ **then** "Yes" **else** "No"
  }

Observaciones: 1. Igual que antes, al reiterar m veces la selección de números de prueba a la probabilidad de error se minimiza, a lo sumo es 2^{-m}.

2. La complejidad del algoritmo, en cuanto a operaciones aritméticas y cálculos de residuos entre números menores que n^2, es

$$O(m \log n).$$

10.2 Máquinas probabilísticas

10.2.1 Máquinas del tipo 1

Una *máquina de Turing probabilística del tipo 1*, (mTp1), es una mTnd M modificada de manera que en su árbol de computaciones:

- cada arista tiene un peso numérico $p \in [0,1]$,
- en cada nodo la suma de los pesos de las aristas que emanan de él es 1,
- el conjunto de nodos terminales es la unión disjunta de nodos de aceptación y de nodos de rechazo,

$$\text{Terminales} = \text{Aceptantes} \cup \text{Rechazantes},$$

y

- toda computación es terminal.

Las transiciones en una mTp1 se suponen independientes de sus precedentes, por lo que la probabilidad de una computación es el producto de las probabilidades de las transiciones que la componen.

Alternativamente, en una mTp1 podemos considerar a su función de transición como una de la forma

$$\begin{array}{rcl} t : Q \times \Sigma \times [1,n] & \rightarrow & Q \times \Sigma \times \{\text{Der,Izq}\} \\ (p,\sigma,r) & \mapsto & (q,\tau,m) \end{array}$$

donde $r : Q \times \Sigma \times R \rightarrow [1,n]$ es una variable aleatoria tal que

$$\text{Prob}\{r(p,\sigma,x) = i\} = \text{Prob}\{\text{pasar al estado } i \text{ al estar en } p \text{ y leer } \sigma\}.$$

Sea M una mTp1. Para cada cadena de entrada $\mathbf{x}$ sea

$$\rho_M(\mathbf{x}) = \text{Prob}\{M \text{ acepta a } \mathbf{x}\}.$$

El lenguaje *reconocido* por M es L si y sólo si

- $\forall \mathbf{x} \quad : \quad \mathbf{x} \notin L \;\Rightarrow\; \rho_M(\mathbf{x}) = 0,$
- $\exists \epsilon > 0 \forall \mathbf{x} \quad : \quad \mathbf{x} \in L \;\Rightarrow\; \rho_M(\mathbf{x}) > \epsilon.$

Escribiremos $L = L(M)$.

Se tiene evidentemente que

- toda máquina de Turing determinística es una mTp1, en la que en todo instante sólo una transición asume una probabilidad 1, y
- toda mTp1 es una máquina de Turing no-determinística, de hecho las nociones de *reconocimiento* coinciden, sólo que en una mTp1 se sabe que una cadena reconocida posee al menos $100 \cdot \epsilon\%$ computaciones aceptantes.

Consideraremos únicamente mTp1's tales que sus árboles de computación poseen alturas polinomiales en la longitud de sus entradas. Hacemos

$$R = \{L | \exists M \in \text{mTp1} : L = L(M)\}.$$

Entonces

$$P \subset R \subset NP.$$

10.2.2 Máquinas-p

Sea $p \in [0,1]$. Una *p-máquina de Turing probabilística*, (p-mTp), es una mTnd M modificada de manera que en su árbol de computaciones:

- cada nodo no terminal posee exactamente dos hijos,
- cada arista tiene uno de dos pesos numéricos $p, 1-p \in [0,1]$, y la suma de los pesos de las aristas que emanan de cada nodo es 1,
- el conjunto de nodos terminales es la unión disjunta de nodos de aceptación y de nodos de rechazo,

$$\text{Terminales} = \text{Aceptantes} \cup \text{Rechazantes},$$

y

- toda computación es terminal.

El lenguaje *reconocido* por M es L si y sólo si

- $\forall \mathbf{x} \quad : \quad \mathbf{x} \in L \;\Rightarrow\; \rho_M(\mathbf{x}) \geq p.$
- $\forall \mathbf{x} \quad : \quad \mathbf{x} \notin L \;\Rightarrow\; \rho_M(\mathbf{x}) \leq 1-p,$

Escribiremos $L = L(M)$.

Observaciones: 1. Con esta noción de reconocimiento, ninguna palabra deja el valor "*quién_sabe*" en una p-mTp.

2. Si $p < 1-p$ entonces $L(M)$ es vacío para cada p-mTp M.

3. Para $p, q \geq \frac{1}{2}$ se tiene que las clases de

- p-mTp's y
- q-mTp's

son equivalentes.

4. Un valor $\frac{1}{2}$ a la probabilidad de reconocimiento de una palabra no da información alguna sobre el reconocimiento de esa palabra.

Sea

$$PP = \{L | \exists M \in \tfrac{1}{2}\text{-mTp's} : L = L(M)\}.$$

Se tiene entonces la inclusión

$$NP \cup \text{co-}NP \subset PP \subset PSPACE.$$

Introduzcamos una noción de reconocimiento más estricta.

El lenguaje *reconocido* por una p-mTp M, con $0 \leq p < 1$, es L si y sólo si

- $\exists \epsilon, \forall \mathbf{x} \quad : \quad \mathbf{x} \in L \;\Rightarrow\; \rho_M(\mathbf{x}) \geq p + \epsilon.$
- $\forall \mathbf{x} \quad : \quad \mathbf{x} \notin L \;\Rightarrow\; \rho_M(\mathbf{x}) \leq 1-p,$

Escribiremos $L = L_B(M)$.

Observaciones: 1. Con esta noción de reconocimiento, todo lenguaje reconocido por una mTp1 lo es también por una $\frac{1}{2}$-mTp.

Sea

$$BPP = \{L | \exists M \in \tfrac{1}{2}\text{-mTp's} : L = L_B(M)\}.$$

Se tiene entonces las inclusiones

$$P \subset R \subset BPP \subset \Sigma_2^p \cap \Pi_2^p.$$

Observaciones: 1. Probar que un problema está en R o en BPP es tan bueno como probarlo en P pues iterando el procedimiento de decisión se puede minimizar tanto cuanto se quiera la probabilidad de error sin aumentar por esto el tiempo polinomial de solución.

2. A la fecha no se conocen problemas completos en las clases R y BPP.

Computabilidad con mTp's

Dada una $\frac{1}{2}$-mTp M para una entrada $x \in \Sigma^*$ a $M(x)$ la podemos considerar como una variable aleatoria

$$M(x) : [1, n]^* \to \Sigma^*.$$

La función *calculada* por M es la función

$$f_M : x \mapsto \begin{cases} y & \text{si Prob}\{M(x) = y\} > \frac{1}{2} \\ \perp & \text{si no existiera tal } y. \end{cases}$$

Recíprocamente, f es *computable probabilísticamente* si existe una $\frac{1}{2}$-mTp M tal que $f = f_M$.

Proposición: Una función es computable probabilísticamente si y sólo si es computable Turing.

Dem: "*si*": Las mT's son casos particulares de mTp's. Luego toda función computable Turing es computable probabilísticamente.

"*sólo si*": Si f es computable probabilísticamente entonces para cada x puede encontrarase algorítmicamente a la y correspondiente tal que $y = f(x)$. Este procedimiento atestigua a f como computable Turing.

Así pues, formalmente, la noción de probabilidad no aumenta la capacidad de cómputo. Las ventajas de considerar probabilidades las dan los tiempos y los espacios de cómputo.

La función de *error* en una $\frac{1}{2}$-mTp M es

$$e_M : x \mapsto \begin{cases} \text{Prob}\{M(x) \neq f_M(x)\} & \text{si } f_M(x) \downarrow, \\ \perp & \text{en otro caso.} \end{cases}$$

Se tiene que para todo x:

$$x \in \text{Dom}(e_M) \;\Rightarrow\; e_M(x) < \frac{1}{2}.$$

Si existe $\epsilon \in]0, \frac{1}{2}[$ tal que

$$x \in \text{Dom}(e_M) \;\Rightarrow\; e_M(x) \leq \epsilon,$$

decimos que M calcula a f_M con una *probabilidad de error acotada.*

Resulta claro que una mTdeterminística es una mTp cuya función de error es nula.

10.2.3 Algunas inclusiones entre clases

Definamos

$$ZPP = R \cap \text{co-}R.$$

Recordamos que un algoritmo en R es de *Monte-Carlo*: Ante respuestas afirmativas contesta afirmativamente, pero ante respuestas negativas puede contestar afirmativamente, aunque con probabilidades mínimas.

Un algoritmo de tipo *Las Vegas* es tal que nunca miente: Ante respuestas afirmativas contesta afirmativamente, pero ante respuestas negativas contesta que no puede probar valores afirmativos.

Los algoritmos de la clase ZPP son del tipo Las Vegas.

Se tiene entonces las inclusiones siguientes:

$$\begin{array}{ccccccccc} P & \subset & ZPP & \subset & R & \subset & NP & & \\ & & & & \cap & & \cap & & \\ & & & & BPP & \subset & PP & \subset & PSPACE \end{array}$$

Capítulo 11

Complejidad de Kolmogorov

11.1 Introducción

La *información* contenida en una cadena de caracteres la podemos representar como la longitud del programa mínimo para construirla.

- 1^n se construye con un programa de longitud $O(\log n)$.
- $\pi = 3.14159\ldots$ se construye con un programa de longitud $O(1)$.

Conforme la cadena dada sea más "regular", tanto más "comprimible" ha de ser: Un programa que la construya será muy compacto.

Así pues la noción de "aleatoriedad" puede identificarse como "no-compresibilidad".

Definición: Una sucesión

$$(a_n)_{n\in N} \in \{0,1\}^{<\omega}$$

es un *colectivo* si

1. La probabilidad de aparición de 1 en la sucesión está bien definida:

$$\exists p \in [0,1] : \lim_{n\to\infty} \frac{1}{n+1}\text{card}\{m < n | a_m = 1\} = p.$$

2. Dada una función $\phi : \{0,1\}^* \to \{0,1\}$ definamos la sucesión

$$\mathbf{s}_\phi = (s_n)_{n\in N}$$

haciendo

$$\begin{aligned} s_0 &= \text{Min}\{m | \phi(a_0 \ldots a_{m-1}) = 1\} \\ s_{n+1} &= \text{Min}\{m | \phi(a_0 \ldots a_{m-1}) = 1 \ \wedge \ m > s_n\} \end{aligned}$$

Pues bien, para la noción de "colectivo" hemos de tener también que para todo ϕ, en una clase de funciones dada, por ejemplo, la clase de funciones recursivas totales,

$$\lim_{n\to\infty} \frac{1}{n+1}\text{card}\{m < n | a_{s_m} = 1\} = p.$$

$\frac{nil}{0}$															
$\frac{0}{1}$	$\frac{1}{2}$														
$\frac{00}{3}$	$\frac{01}{4}$	$\frac{10}{5}$	$\frac{11}{6}$												
$\frac{000}{7}$	$\frac{001}{8}$	$\frac{010}{9}$	$\frac{011}{10}$	$\frac{100}{11}$	$\frac{101}{12}$	$\frac{110}{13}$	$\frac{111}{14}$								
$\frac{0000}{15}$	$\frac{0001}{16}$	$\frac{0010}{17}$	$\frac{0011}{18}$	$\frac{0100}{19}$	$\frac{0101}{20}$	$\frac{0110}{21}$	$\frac{0111}{22}$	$\frac{1000}{23}$	$\frac{1001}{24}$	$\frac{1010}{25}$	$\frac{1011}{26}$	$\frac{1100}{27}$	$\frac{1101}{28}$	$\frac{1110}{29}$	$\frac{1111}{30}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figura 11.1: Correspondencia $N \to \{0,1\}^{<\omega}$.

Ejemplo: Sea $A = (a_n)_{n\in N} \in \{0,1\}^{<\omega}$ una sucesión tal que satisface la primera condición anterior:

$$\exists p \in [0,1] : \lim_{n\to\infty} \frac{1}{n+1}\text{card}\{m < n | a_m = 1\} = p.$$

Consideremos la función

$$\phi_1 : (a_0 \dots a_{n-1}) \mapsto \begin{cases} 1 & \text{si } a_n = 1, \\ \perp & \text{en otro caso.} \end{cases}$$

Si se cumpliera la segunda condición, hemos de tener $p = 1$.

Sin embargo para la función

$$\phi_2 : (a_0 \dots a_{n-1}) \mapsto \neg a_n,$$

hemos de tener que $p = 0$.

Así pues en clases grandes de funciones la definición anterior no se puede satisfacer.

11.2 Presentación de la teoría

Identificaremos a N con $\{0,1\}^{<\omega}$ mediante la correspondencia

$$\begin{array}{rcl} x : N & \to & \{0,1\}^{<\omega} \\ n & \mapsto & x(n) \end{array}$$

donde

$$x(n) = \begin{cases} nil & \text{si } n = 0, \\ (n - 2^k + 1)_{2,k} & \text{si } n > 0, \text{ con } k = \lfloor \log_2(n+1) \rfloor. \end{cases}$$

($(n)_{2,k}$ es la representación en base 2 de n utilizando exactamente k bits).

En la Figura 1 presentamos esta correspondencia. Cada entrada es de la forma $\frac{x(n)}{n}$.

$x(n)$ es pues una cadena de longitud $O(\log(n))$.

La función inversa es

$$\begin{array}{rcl} n : \{0,1\}^{<\omega} & \to & N \\ x & \mapsto & n(x) \end{array}$$

Denotemos por $|x|$ a la longitud de la cadena x. Resulta claro que

$$n(x) = 2^{|x|} - 1 + (x)_2.$$

Esta última función es un *programa universal* S que dado el código p, como una cadena en $\{0,1\}^{<\omega}$, da el número n.

La *complejidad de Kolmogorov* de una cadena x es la longitud del mínimo *programa* p que reconstruye a x:

$$K_S(x) = \text{Min}\{|p| : S(p) = n(x)\}.$$

Definiciones básicas

$$\begin{array}{rcl} D & : & \text{Un conjunto.} \\ \left.\begin{array}{rcl} n : D & \to & N \\ x & \mapsto & n(x) \end{array}\right\} & : & \text{Una enumeración.} \\ \left.\begin{array}{rcl} S : \{0,1\}^{<\omega} & \to & N \\ p & \mapsto & n = S(p) \end{array}\right\} & : & \text{Procedimiento generador de } \textit{objetos}. \end{array}$$

Complejidad de Kolmogorov: $\forall x \in D$:

$$K_S(x) = \begin{cases} \text{Min}\{|p| : S(p) = n(x)\} & \text{si existe tal } p, \\ \infty & \text{en otro caso.} \end{cases}$$

Observación

Si $(S_i)_{i\in[1,r]}$ es una colección de métodos para generar a D entonces se puede construir un método S tal que

$$\exists c(< \log(r))\ \forall x \in D :\ K_S(x) \leq \underset{i\leq r}{\text{Min}}\{K_{S_i}(x)\} + c.$$

En efecto, para obtener S lo que hay que hacer es, a cada programa p que genere a x, adjuntarle una cadena de $\log(r)$ bits que indique el procedimiento S_i a elegir entre los r diponibles.

Dados dos procedimientos S, T se dice que S *absorbe* a T si

$$\exists c :\ K_S(x) \leq K_T(x) + c\ \forall x.$$

Los dos métodos son *c-equivalentes* si uno absorbe al otro con precisión c, es decir,

$$|K_S(x) - K_T(x)| \leq c.$$

Observaciones

1. En la clase de funciones computables-Turing existe una función S que absorbe a todas:

   $$\forall T \in \{\text{Recursivas}\}\ \exists c_{S,T} :\ K_S(x) \leq K_T(x) + c_{S,T}\ \forall x.$$

   Tal S corresponde a una función universal. Así pues, las funciones universales son *óptimas*.

2. Cualesquiera dos funciones óptimas son equivalentes, es decir, si S, T son dos funciones óptimas

   $$\exists c_{S,T} :\ |K_S(x) - K_T(x)| \leq c_{S,T}.$$

11.2.1 Complejidades condicionales

Un *intérprete* en D es una función

$$\begin{array}{rcl} f : \{0,1\}^{<\omega} \times D & \to & D \\ (p, y) & \mapsto & f(p, y) = x \end{array}$$

Para cada $x \in D$ su *complejidad condicional*, dada y según f, es

$$K_f(x|y) = \begin{cases} \text{Min}\{|p| : f(p, y) = x\} & \text{si existe tal } p, \\ \infty & \text{en otro caso.} \end{cases}$$

Teorema de Invarianza

Existe una función recursiva f_0 tal que para cualquier otra función recursiva f existe una constante c_f de manera que para cualesquiera dos objetos x, y se tiene

$$K_{f_0}(x|y) \leq K_f(x|y) + c_f.$$

- f_0 es un intérprete mínimo que se construye mediante una máquina universal.
- Dos funciones óptimas son equivalentes.

Así pues al fijar una máquina universal U_0 como máquina de *referencia* definimos:

- *Complejidad condicional* de x dado y: $K(x|y) = K_{f_0}(x|y)$.
- *Complejidad incondicional* de x: $K(x) = K(x|nil)$.

Ejemplos

1. $K(xx) \leq K(x) + O(1)$.

2. Sea $< \cdot >: D \times D \to D$ una función de apareamiento. Para una pareja $(x, y) \in D^2$ definamos

$$K(x, y) = K(< x, y >).$$

Como es necesario mantener un registro de las longitudes de x y de y tendremos

$$K(x, y) \leq K(x) + K(y) + O(\log(\text{Min}\{K(x), K(y)\})).$$

11.2.2 Incomprimibilidad

Como una consecuencia del Teorema de Invarianza se tiene que para alguna constante c:

$$K(x) \leq n(x) + c.$$

Ahora bien,puesto que hay 2^n cadenas de longitud n y hay a lo sumo $2^n - 1$ descripciones de longitud menor que n necesariamente alguna cadena de longitud n ha de tener una descripción de longitud al menos de n. Tal cadena es incomprimible. Formalmente:

x es *incomprimible* si

$$K(x) \geq |x|.$$

Similarmente podemos ver que para todos n, y existe un objeto x tal que

$$K(x|y) \geq n(x).$$

Ejemplos y observaciones

1. Sea x de la forma $x = uvw$, donde v es una partícula comprimible,

$$K(v) < |v|.$$

Entonces v no puede comprimirse demasiado.

En efecto, a x lo podemos recosntruir mediante

- una descripción de $|u|$,
- una descripción de uw,
- separadores de las dos descripciones anteriores.

Luego

$$K(x) \leq K(v) + O(\log |x|) + |uw|.$$

Como $|x| \leq K(x)$ se tiene

$$K(v) \geq |v| - O(\log |x|).$$

Una cadena incomprimible puede tener cadenas comprimibles (pero no tanto). Esto es similar al hecho de que una cadena aleatoria de 0's y 1's contiene cadenas constantes de longitud arbitraria.

2. Para cada objeto x sea $p(x)$ un programa mínimo que genera a x. Entonces $p(x)$ es en sí incomprimible. De hecho existe c tal que

$$K(p(x)) \geq |p(x)| - c \; forall x.$$

En efecto, supongamos lo contrario. Entonces

$$\forall x \; \exists c_x : \; K(p(x)) \leq |p(x)| - c_x.$$

Sea U una máquina universal de referencia y sea $V = U^2$. Entonces

$$\forall x : \; V(p(p(x))) = x.$$

Luego

$$\begin{array}{rcl} K(x) & = & K(p(p(x)) \\ & \leq & |p(p(x))| - c_{p(x)} \\ & \leq & |p(x)| - c_{x,*} + n(V) + 1 \end{array}$$

lo cual es absurdo pues necesariamente $K(x) = |p(x)|$.

Sea $g : N \to N$. Diremos que una cadena x es *g-incomprimible* si

$$K(x) \geq n(x) - g(n(x)).$$

Para cada n, la razón entre los objetos comprimibles respecto a todos los de longitud n es

$$\frac{2^{n-g(n)} - 1}{2^n} \approx 2^{-g(n)}$$

lo que converge a cero si g es creciente.

Los objetos log-comprimibles se dicen ser *aleatorios*.

Si x es una sucesión de longitud infinita, x es *g-incomprimible* si para cada n es segmento inicial de longitud n lo es:

$$\forall n : K(x[0, n-1]) \geq n - g(n).$$

11.2.3 Descripciones auto-delimitadoras

Consideremos la función

$$\bar{\cdot} : \{0,1\}^{<\omega} \to \{0,1\}^{<\omega}$$

definida inductivamente como sigue:

$$\begin{array}{rcl} x \in \{0,1\} & \Rightarrow & \overline{x} = x1 \\ x = ay & \Rightarrow & \overline{x} = a0\overline{y} \\ a \in \{0,1\} & & \\ y \in \{0,1\}^{<\omega} & & \end{array}$$

$\overline{x}$ se obtiene de intercalar un 0 entre cualesquiera dos símbolos de x y finalizar con un 1.

La cadena $\pi(x) = \overline{|x|} \cdot x$ se dice ser la versión *autodelimitadora* de x. Si la longitud de x es n la longitud de $\pi(x)$ es $n + 2\log(n)$.

Una cadena de cadenas $\mathbf{x} = x_1 \cdots x_k$ se codifica yuxtaponiendo a los códifos:

$$\pi(\mathbf{x}) = \pi(x_1) \cdot \cdots \cdot \pi(x_k).$$

11.3 Estimaciones de la complejidad

11.3.1 Kolmogorov

Sea $A \in N \times N$ recursivamente enumerable. Para cada $y \in N$ sea

$$M_y = \{x | (x,y) \in A\} = A^{-1}(y).$$

Entonces, salvo una constante que depende de A se tiene $\forall x, y$:

$$K(x|y) \leq |\mathrm{card}(M_y)|.$$

Conjuntos ralos

Sea A un conjunto. Definamos

$$A^{\leq n} = \{x \in A | |x| \leq n\}.$$

A es *ralo* si

$$\lim_{n\to\infty} \frac{1}{n} \mathrm{card}(A^{\leq n}) = 0.$$

Lema 1: Si A es un conjunto recursivo y ralo entonces para cada constante c se tiene que sólo para un número finito de elementos x se cumple

$$K(x) \geq |x| - c.$$

Lema 2: Sea A es un conjunto recursivo enumerable y $\epsilon > 0$. Si

$$\lim_{n\to\infty} \frac{1}{n^{-(1+\epsilon)2^n}} \mathrm{card}(A^{\leq n}) = 0$$

entonces para cada constante c se tiene que sólo para un número finito de elementos x se cumple

$$K(x) \geq |x| - c.$$

Lema 3: Sea A es un conjunto recursivo enumerable. Si

$$\text{card}(A^{\leq n}) \leq p(n)$$

donde $p(X)$ es un polinomio, entonces para cada constante c se tiene que sólo para un número finito de elementos x se cumple

$$K(x) \geq \frac{|x|}{c}.$$

11.3.2 Propiedades

Para las cadenas x se cumple:

1. $K(x) \leq |x|$, salvo una constante independiente de x.

2. La relación anterior se cumple para la mayoría de las palabras:

$$frac\text{card}\{x : K(x) < |x| - m\}2^m \leq 2^{-m}.$$

3. $\lim_{|x|\to\infty} K(x) = \infty$.

4. Más aún:
Si $m(x) = \text{Min}\{K(y)|y \geq x\}$ entonces $\lim_{|x|\to\infty} m(x) = \infty$.

5. La función $m(x)$ queda a la larga dominada por una función recursiva monótona creciente.

6. K es "uniformemente contínua":

$$|K(x+h) - K(x)| \leq 2|h|.$$

Capítulo 12

Trabajos a Desarrollar

12.1 Primera lista de ejercicios

1. Pruebe que no existen dos enteros p, q tales que $\left(\frac{p}{q}\right)^2 = 3$.

2. Demuestre que si $x \in \{a, b\}^*$ es tal que $abx = xab$ entonces existe n tal que $x = (ab)^n$.

3. Encuentre la falacia de la siguiente "prueba" de que cualesquiera dos números $x, y \in N$ son iguales.

DEM. Sea

$$P(n) \equiv (\forall x)(\forall y)[\text{Max}(x, y) = n \ \Rightarrow \ x = y].$$

Evidentemente $P(0)$ es verdadera.

Ahora, suponga que se cumple $P(n)$. Veamos que se ha de cumplir $P(n+1)$. Si $\text{Max}(x, y) = n + 1$ entonces $\text{Max}(x_1, y_1) = n$, donde $x_1 = x - 1$ y $y_1 = y - 1$. Por la hipótesis de inducción se ha de tener $x_1 = y_1$. Luego $x = y$, por lo que $P(n+1)$ es cierto. □

4. Encuentre la falacia de la siguiente "prueba" de que

De noche todos los gatos son de un mismo color, digamos pardo.

DEM. Sea $GN =$ {gatos que andan de noche}.

Probemos por inducción en $n = \text{card}(GN)$ que cualesquiera dos elementos en GN son de un mismo color.

Si $n = 1$ la aseveración es correcta.

Supongamos que hay $n + 1$ gatos. Quitemos uno, que llamamos *Yago*. Los restantes son n y por la hipótesis de inducción son todos del mismo color. Devolvemos a Yago y quitamos otro, digamos *Prudence*. Quedan n y por tanto todos ellos son del mismo color. Yago y Prudence han de ser del mismo color y consecuentemente *todos* los gatos son del mismo color. □

5. Estime el número de bits necesarios para almacenar el código de Cantor de una sucesión de m números enteros, cada uno de los cuales se escribe con a lo sumo n bits.

6. Decida cuáles de los siguientes conjuntos son numerables. Justifique su respuesta:

a) El conjunto de máquinas de Turing.

b) El conjunto de números complejos.

c) El conjunto de números racionales.

d) El conjunto de intervalos en los reales, abiertos o cerrados, con extremos racionales.

e) El conjunto de funciones INYECTIVAS de $\mathbb{N}$ en $\mathbb{N}$.

f) El conjunto de polinomios con coeficientes racionales.

h) El conjunto de programas en C.

7. Un programa-**while** sin instrucciones **while** se dice ser un programa *rectilíneo*.

a) Muestre que todo programa-**while** que posee exactamente una variable es equivalente a un programa rectilíneo, en el sentido de que ambos calculan a la misma función.

b) Muestre que toda función calculada por un programa rectilíneo es total.

c) Construya un programa-**while** con exactamente dos variables que no sea equivalente a programa rectilíneo alguno.

Sugerencia: Utilice lo mostrado en el inciso b).

8. Escriba programas-**while** para calcular cada una de las siguientes funciones:

$$\begin{array}{l} z := x * y \\ z := x \ \mathbf{mod}\ y \\ z := x * *y \quad , \text{ considere } 0 * *0 = 0, \\ z := 2^x \end{array}$$

9. Escriba un programa-**while** que calcule a la función $z := x - -$ en términos de las funciones $z := 0$ y $z := x + +$.

10. Muestre que se obtiene exactamente a la misma clase de programas-**while** si sustituímos las pruebas de la forma $x \neq 0$ por las de la forma $x \neq y$.

11. Muestre que ningún programa-**while** de una sola variable puede calcular a la función $x \mapsto 2 * x$.

12. Clasifique a las funciones $N \to N$ que se calculan por

a) los programas rectilíneos de una sola variable.

b) los programas rectilíneos con cualquier número de variables.

13. a) Muestre que la función $\mathit{RaizEnt} : N \to N$ tal que $\forall x : \mathit{RaizEnt}(x) = \lfloor \sqrt{x} \rfloor$ es computable.

b) Muestre que el *exceso de cuadrado* $\mathit{ExCua} : x \mapsto x \dot{-} [\mathit{RaizEnt}(x)]^2$ es computable.

14. Muestre que las siguientes funciones son computables:

a) $f_1 : x \mapsto \begin{cases} 1 & \text{si } x \text{ es par,} \\ 0 & \text{en otro caso,} \end{cases}$

b) $f_2 : x \mapsto \begin{cases} 1 & \text{si } x \text{ es par,} \\ \perp & \text{en otro caso,} \end{cases}$

15. a) Muestre que las funciones computables son computables, también, por máquinas de Turing.

b) Muestre que las funciones computables por máquinas de Turing son, también, computables por programas-**while**.

16. Muestre que las siguientes funciones son computables

$$\begin{aligned}
Div(x,y) &= \begin{cases} 1 & \text{si } x \text{ divide a } y, \\ 0 & \text{en otro caso,} \end{cases} \\
pr(x) &= x\text{-ésimo número primo,} \quad primo(0) = 2 \\
primo(x) &= \begin{cases} 1 & \text{si } x \text{ es primo,} \\ 0 & \text{en otro caso,} \end{cases} \\
exp(x,y) &= \text{exponente del } x\text{-ésimo número primo en la descomposición en primos de } y, \\
&\quad \text{por ejemplo, para}
\end{aligned}$$

$$y = 16765056000 = 2^{10}3^5 5^3 7^2 11,$$

se tiene

x	$exp(x,y)$
0	10
1	5
2	3
3	2
4	1
≥ 5	0

17. Muestre que si $g, f_1, \ldots, f_k, f$ son funciones computables por máquinas de Turing, entonces las funciones $Comp(g; f_1, \ldots, f_k)$, $Mini(f)$ y $MiAc(f)$ son computables también por máquinas de Turing.

18. Muestre que si g, h, c son computables por máquinas de Turing, entonces las funciones $Recu(g; h)$ y $ReAc(g; h; c)$ son computables también por máquinas de Turing.

19. Muestre que si en el esquema de minización se omite la exigencia de que la función f sobre la que se aplica sea total entonces habría funciones computables cuya minimización no lo sería.

20. Para una función $f : N \to N$ definamos las funciones siguientes:

$$\begin{aligned}
y \mapsto Af(y) &= \begin{cases} \text{Min}\{x | f(x) = y\} & \text{si existe tal } x, \\ 1 & \text{en otro caso.} \end{cases} \\
x \mapsto Bf(x) &= \begin{cases} 0 & \text{si } f(x) = 0, \\ 1 & \text{si } f(x)\downarrow \wedge f(x) \geq 1, \\ \perp & \text{en otro caso.} \end{cases} \\
x \mapsto Cf(x) &= \begin{cases} \lfloor \log_2 f(x) \rfloor & \text{si } f(x)\downarrow, \\ \perp & \text{en otro caso.} \end{cases}
\end{aligned}$$

a) Muestre que la clase de funciones computables de un solo argumento son cerradas bajo los operadores B y C.

b) Muestre que la clase de funciones computables de un solo argumento que además son crecientes y tienen una imagen infinita son cerradas bajo A pero no bajo B y C.

12.2 Primera lista de programas

En esta sección, al construir un programa-**while** hay que utilizar variables de la forma Xw, donde $w \in \{0,1\}^*$ es una palabra que señala el orden de aparición de la variable.

1. *Expansor de macros condicionales*: Escriba un programa que traduzca macros "programáticos".

Entrada: Un programa con instucciones **if -then** , **if -then -else** , **repeat -until** .

Salida: El programa-**while** equivalente.

2. *Expansor de macros de pruebas complejas*: Escriba un programa que traduzca pruebas compuestas.

Entrada: Una prueba compuesta, es decir, una prueba escrita como una composición booleana de pruebas básicas, involucrando un número cualquiera de variables:

$$\Phi(X_1, \ldots, X_m).$$

Salida: Una prueba básica, es decir, de la forma $X \neq 0$, equivalente a la dada:

$$\Phi(X_1, \ldots, X_m) \Leftrightarrow X \neq 0.$$

3. *Expansor de macros de expresiones aritméticas*: Escriba un programa que reescriba expresiones aritméticas como programas-**while**.

Las expresiones aritméticas se forman con constantes y variables e involucran a las cuatro operaciones aritméticas, a la exponenciación, **div**, **mod**, **sqrt**, parte entera, entero más próximo por arriba, etc.

Entrada: Una asignación cuyo miembro a la derecha es una expresión aritmética:

$$X := T(X_1, \ldots, X_m).$$

Salida: Un programa-**while** equivalente al anterior.

4. *Esquema de recursión*: Escriba un programa que aplique el esquema de recursión.

Entrada: Dos programas-**while** G y H, vistos como programas que calculan a las funciones

$$\begin{aligned} g : X &\mapsto X_{n_g} \\ h : (X, X0) &\mapsto X_{n_h}, \end{aligned}$$

donde X_{n_g} y X_{n_h} son las últimas variables de G y H respectivamente.

Salida: Un programa-**while** que calcula a la función f tal que

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

5. *Compresión sintáctica de programas*: Considerando la codificación de símbolos mostrada en la tabla 12.1, escriba a cualquier programa-**while** como una sucesión de bytes.

6. *Decompresión sintáctica de programas*: Para la misma codificación de símbolos en el programa anterior, decida cuándo una sucesión de bytes es el código de un programa-**while**. Cuando lo sea, reescriba el programa como una cadena de caracteres ASCII.

7. *Simulador de programas*-**while**: Dado un programa-**while** y una configuración inicial a su lista de variables, aplicar el programa a esa configuración y visualizar el cómputo paso a paso.

Entrada: Un programa-**while** P y una instancia inicial $\mathbf{x}$ a su lista de variables.

Salida: Una visualización del cómputo de $P(\mathbf{x})$.

Símbolo	Código hexadecimal
while	0
do	1
{	2
}	3
++	4
–	5
$\neq$	6
X	7
0	8
1	9
;	A

Tabla 12.1: Codificación de símbolos de los programas-**while**.

8. *Traductor de máquinas de Turing a programas*-**while**: Dada una máquina de Turing escriba el programa-**while** que calcula a la misma función calculada por la máquina de Turing.

Aquí hay que suponer que la máquina de Turing está definida sobre el alfabeto $\{0, 1\}$, que el 0 hace el papel de "blanco" y que se representa a los números en unario.

9. *Convertidor de máquinas de Turing a máquinas de Turing con una cinta semi-infinita*: Dada una máquina de Turing $M \in \mathcal{MT}$ escriba una máquina de Turing $M' \in \mathcal{MTSI}$ que sea equivalente a M pero sólo ocupe una cinta semi-infinita, es decir, a partir de una casilla fija, se puede extender todo cuanto sea necesario a la derecha de esa casilla. Describa la función de transformación de instancias.

10. *Convertidor de máquinas de Turing de n-pistas a máquinas de Turing*: Dada una máquina de Turing $M \in \mathcal{MTVP}$ con n pistas, escriba una máquina de Turing $M' \in \mathcal{MT}$ que sea equivalente a M. Describa la función de transformación de instancias.

Toda $M \in \mathcal{MTVP}$ con n pistas puede ser dada mediante un arreglo

$$T_M : \textbf{ array } Q \times A^n \textbf{ of } Q \times A^n \times \{Izq^n, Der^n\}$$

donde A es el alfabeto de la máquina y Q es el conjunto de estados.

Toda $M' \in \mathcal{MT}$ (a secas) puede ser dada mediante un arreglo

$$T_{M'} : \textbf{ array } Q \times A \textbf{ of } Q \times A \times \{Izq, Der\}$$

Entrada: El arreglo T_M que determina a la máquina $M \in \mathcal{MTVP}$.

Salida: El arreglo $T_{M'}$ que determina a la máquina $M' \in \mathcal{MT}$ equivalente a M y la función ϕ que transforma palabras en el alfabeto de M en palabras en el alfabeto de M'.

11. *Convertidor de máquinas de Turing de n-cintas a máquinas de Turing*: Dada una máquina de Turing $M \in \mathcal{MTVC}$ con n cintas, escriba una máquina de Turing $M' \in \mathcal{MT}$ que sea equivalente a M. Describa la función de transformación de instancias.

Toda $M \in \mathcal{MTVC}$ con n cintas puede ser dada mediante un arreglo

$$T_M : \textbf{ array } Q \times A^n \textbf{ of } Q \times A^n \times \{Izq, Der\}^n$$

donde A es el alfabeto de la máquina y Q es el conjunto de estados.

Entrada: El arreglo T_M que determina a la máquina $M \in \mathcal{MTVC}$.

Salida: El arreglo $T_{M'}$ que determina a la máquina $M' \in \mathcal{MT}$ equivalente a M y la función ϕ que transforma palabras en el alfabeto de M en palabras en el alfabeto de M'.

12. *Convertidor de máquinas de Turing de n-cintas a máquinas de Turing*: Dada una máquina de Turing $M \in \mathcal{MTVC}$ con n cintas, escriba una máquina de Turing $M' \in \mathcal{MT}$ que sea equivalente a M. Describa la función de transformación de instancias.

Toda $M \in \mathcal{MTVC}$ con n cintas puede ser dada mediante un arreglo

$$T_M : \textbf{ array } Q \times A^n \textbf{ of } Q \times A^n \times \{Izq, Der\}^n$$

donde A es el alfabeto de la máquina y Q es el conjunto de estados.

Entrada: El arreglo T_M que determina a la máquina $M \in \mathcal{MTVC}$.

Salida: El arreglo $T_{M'}$ que determina a la máquina $M' \in \mathcal{MT}$ equivalente a M y la función ϕ que transforma palabras en el alfabeto de M en palabras en el alfabeto de M'.

13. *Convertidor de máquinas de Turing a máquinas de Turing sobre el alfabeto* $(0+1)$: Dada una máquina de Turing $M \in \mathcal{MT}$ sobre un alfabeto A arbitrario, escriba una máquina de Turing $M' \in \mathcal{MT}01$ que sea equivalente a M. Describa la función de transformación de instancias.

12.3 Segunda lista de ejercicios

1. Describa una función F : *programas*-**while**'*s* $\to \mathbb{N}$ que sea efectivamente una biyección.

2. *Códigos basados en primos*: Consideremos a los primeros 7 números primos: 2, 3, 5, 7, 11, 13 y 17, y a la codificación de los elementos sintácticos de los programas-**while**'s $\ulcorner\cdot\urcorner$: *programas*-**while**'*s* $\to \mathbb{N}$ construída como sigue:

Representación de variables: A la variable Xw, donde $w \in \{0,1\}$ es una lista de 0's y 1's, la representamos por el número entero

$$r(Xw) = 2^{long(w)} + (w)_2.$$

Codificación de instrucciones:

$$\begin{aligned} [Xw++] &\mapsto 17^{r(Xw)} \\ [Xw--] &\mapsto 13^{r(Xw)} \\ [\textbf{while} Xw \neq 0 \textbf{ do } \mathrm{P}] &\mapsto 11^{r(Xw)} \cdot 7^{\ulcorner P \urcorner} \end{aligned}$$

Codificación de programas:

$$\begin{aligned} [Inst; Ruti] &\mapsto 5^{\ulcorner Inst \urcorner} \cdot 3^{\ulcorner Ruti \urcorner} \\ [\{Ruti\}] &\mapsto 2^{\ulcorner Ruti \urcorner} \end{aligned}$$

a) Muestre que $\ulcorner\cdot\urcorner$ es una función inyectiva.

b) Describa a un procedimiento para calcular su inversa.

c) Decida si acaso es una biyección.

3. Considerando a la codificación con primos del ejercicio anterior, describa un procedimiento para calcular los códigos de los numerales.

4. Los códigos basados en primos crecen (excesivamente) rápido. Considerando a la biyección de Cantor describa una codificación similar a la basada en primos con un (mucho) menor crecimiento.

5. Muestre que la función

$$\begin{array}{rcl} f: 0 & \mapsto & 0 \\ 1 & \mapsto & 1 \\ 2 & \mapsto & 2^2 \\ 3 & \mapsto & 3^{3^3} \\ 4 & \mapsto & 4^{4^{4^4}} \\ \vdots & \vdots & \vdots \\ n & \mapsto & \text{pila de } n \text{ } n\text{'s apiladas en exponenciación consecutiva,} \\ \vdots & \vdots & \vdots \end{array}$$

es recursiva primitiva.

6. Muestre que la función

$$f: x \mapsto \begin{cases} 2x & \text{si } x \text{ es un cuadrado perfecto,} \\ 2x+1 & \text{en otro caso,} \end{cases}$$

es recursiva primitiva.

7. Muestre que la función

$$\phi: x \mapsto \text{card}\{y \leq x : y|x\} = (\text{número de divisores de } x),$$

es recursiva primitiva.

Tenemos, por ejemplo, que para $x = 1995 = 3 \cdot 5 \cdot 7 \cdot 19$,

$$\pi(1995) = 16 = 2^4.$$

8. Muestre que la función

$$\sigma: x \mapsto \begin{cases} 0 & \text{si } x = 0 \\ \sum_{y|x} y & \text{en otro caso,} \end{cases}$$

es recursiva primitiva.

$\sigma(x)$ es la suma de divisores de x. Tenemos, por ejemplo, que para

$$x = 1995 = 3 \cdot 5 \cdot 7 \cdot 19,$$

$$\begin{array}{rcl} \sigma(1995) & = & 1+3+5+7+19+15+21+57+35+95+133+ \\ & & 105+285+399+665+1995 \\ & = & 3840. \end{array}$$

9. Muestre que la función

$$\lfloor\sqrt{\cdot}\rceil : x \mapsto n, \text{ donde } n^2 \leq x < (n+1)^2,$$

es recursiva primitiva.

10. Muestre que la función $f : N \to \mathbb{N}$ tal que

$$\begin{aligned} f(0) &= 1, \\ f(1) &= 4, \\ f(2) &= 6, \\ f(x+3) &= f(x) + f(x+1)^2 + f(x+2)^3, \end{aligned}$$

es recursiva primitiva.

11. Sea

$$u : n \mapsto \text{el } n\text{-ésimo entero que es suma de dos cuadrados,}$$

$$\begin{array}{rcl} u : 0 & \mapsto & 0 = 0^2 + 0^2 \\ 1 & \mapsto & 1 = 0^2 + 1^2 \\ 2 & \mapsto & 2 = 1^2 + 1^2 \\ 3 & \mapsto & 4 = 0^2 + 2^2 \\ 4 & \mapsto & 5 = 1^2 + 2^2 \\ 5 & \mapsto & 8 = 2^2 + 2^2 \\ \vdots & \vdots & \vdots \end{array}$$

Muestre que u es recursiva primitiva.

12. Muestre por doble inducción las propiedades siguientes:

a) $\forall n > 0 : f_n(x, y) < f_n(x, y + 1)$.

b) $\forall n > 0 : f_n(x, y) < f_{n+1}(x, y + 1)$.

13. Muestre que $\forall n : \mathcal{E}^2 \subset \mathcal{E}^n$.

14. a) Describa un procedimiento para calcular la función *exponenciación*: $(n, x) \mapsto \begin{cases} x^n & \text{si } x \neq 0 \text{ o } n \neq 0, \\ 0 & \text{en otro caso.} \end{cases}$.

b) Describa un procedimiento más eficiente que el anterior para calcular a la exponenciación, o pruebe que el anterior no se puede mejorar.

Sugerencia: Para calcular x^{31} calcule la sucesión

$$x, x^2, x^8, x^{16}, x^{24}, x^{28}, x^{30}, x^{31}.$$

15. Un número real $x \in [0, 1]$ es *computable* si la función

$$\begin{array}{rcl} f_x : \mathbb{N} & \to & [\![0, 9]\!] \\ n & \mapsto & n\text{-ésimo dígito en la expansión decimal de } x, \end{array}$$

es computable.

Muestre que todo número racional es computable.

16. Muestre que la raíz cuadrada de un número computable es computable.

17. Muestre mediante un argumento "diagonal" que existen números en el intervalo $[0, 1]$ que no son computables.

18. Muestre que π es computable.

Sugerencia: Puede utilizar cualquiera de las siguientes identidades:

$$\frac{\pi}{4} = \sum_{m=0}^{+\infty} \frac{(-1)^m}{2m+1} \qquad\qquad \frac{\pi^2}{6} = \sum_{m=1}^{+\infty} \frac{1}{m^2}.$$

19. Sea $a : \mathbb{N}^2 \to \mathbb{N}$ una función de apareamiento y sean

$$car^a, cdr^a : \mathbb{N} \to \mathbb{N}$$

sus correspondientes funciones proyecciones.

Sea $a^* : \mathbb{N}^* \to \mathbb{N}$ la correspondiente enumeración de las sucesiones de naturales de longitud finita.

Muestre que las siguientes funciones $\mathbb{N}^* \times \mathbb{N}^* \to \mathbb{N}$ son computables:

$$pre(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{si } \mathbf{x} \text{ es prefijo de } \mathbf{y}, \\ 0 & \text{en otro caso.} \end{cases}$$
$$suf(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{si } \mathbf{x} \text{ es sufijo de } \mathbf{y}, \\ 0 & \text{en otro caso.} \end{cases}$$

20. Muestre que las siguientes funciones $\mathbb{N} \times \mathbb{N}^* \to \mathbb{N}^*$ son computables:

$$bor(n, \mathbf{x}) = \begin{cases} \text{prefijo de } \mathbf{x} \text{ de longitud } n & \text{si } n \leq \text{long}(\mathbf{x}), \\ nil & \text{en otro caso.} \end{cases}$$
$$rot(n, \mathbf{x}) = \text{rotación en } n \text{ posiciones de los caracteres en } \mathbf{x}$$

12.4 Segunda lista de programas

1. *Aritmética de grandes números*: Construir un módulo para el manejo de números enteros grandes (número de dígitos ≥ 100). Represente números grandes como listas de enteros sin signos. Construya la suma de enteros grandes procediendo según la intuición más elemental, llevando acarreos.

Construya el producto de enteros grandes procediendo según el algoritmo "divide y vencerás" del libro de Ullman: "Design and analysis of algorithms".

2. *Función de Ackerman*: Implementar el cálculo, mediante "pilas" de la función de Ackerman A. Contar el número de accesos a la pila hechos durante el cálculo de $A(m, n)$.

3. *Función β de Gödel*: Implementar el cálculo de la función β de Gödel. Graficarla en diferentes intervalos. Calcular su inversa mediante el Teorema Chino del residuo.

4. *Minivirus en "C"*: Implementar los programas que se autorreproducen que aparecen en los paneles 5 y 7 del artículo de Witten: "Computer (In)security: Infiltrating Open Systems", ABACUS, Summer 1987, pp: 6-25.

5. *Función de Ackerman*: Implementar el cálculo, mediante el algoritmo cuyo seudocódigo se dió en la Nota 3 de la Parte I de las notas del curso, de la función de Ackerman A. Justificar su funcionamiento.

6. *Códigos de Máquinas de Turing*: Implementar la codificación de máquinas de Turing vistaen clase. Dada una máquina de Turing por su lista de quintuplas generar su código.

7. *Máquina Universal de Turing*: Generar una máquina universal de Turing según el método visto en clase.

8. *Recopilación de un intérprete de programas-***while***'s*: Recolectar los programas elaborados en los puntos 1.1-1.4,1.7 de esta lista y recopilarlos en un solo sistema que sea un intérprete de programas-**while**'s.

12.5 Tercera lista de ejercicios

En esta lista usaremos las notaciones siguientes

$\{f_i\}_{i \geq 0}$: enumeración efectiva de las funciones recursivas.

$\left\{f_i^{(n)}\right\}_{i\geq 0}$: enumeración efectiva de las funciones recursivas cuyo dominio es $\mathbb{N}^n$.

Dada una clase $\mathcal{F}^{(n)}$ de funciones $\mathbb{N}^n \to \mathbb{N}$, una función *universal* para $\mathcal{F}^{(n)}$ es una función $U^{(n)} : \mathbb{N}^{1+n} \to \mathbb{N}$ tal que

- Toda sección de U está en la clase $\mathcal{F}^{(n)}$, i.e.

$$\forall m \in \mathbb{N} : \ (\mathbf{x} \mapsto U(m, \mathbf{x})) = \lambda \mathbf{x}(U(m, \mathbf{x})) \in \mathcal{F}^{(n)}.$$

- Toda función en la clase se "realiza" como una sección de U:

$$\forall f \in \mathcal{F}^{(n)} \exists i_f \in \mathbb{N} : \forall \mathbf{x} \in \mathbb{N}^n \ \ U(i_f, \mathbf{x}) = f(\mathbf{x}).$$

  En tal caso i_f es un *índice* de f correspondiente a U.

Sea $a : \mathbb{N}^2 \to \mathbb{N}$ una función de apareamiento, sea $\mathcal{F}^{(1)}$ una clase de funciones $\mathbb{N} \to \mathbb{N}$ y sea $U^{(1)} : \mathbb{N}^2 \to \mathbb{N}$ una función universal para $\mathcal{F}^{(1)}$. Diremos que $\mathcal{F}^{(1)}$ *contiene a su propia función universal* si $U^{(1)} \circ a \in \mathcal{F}^{(1)}$.

1. Resuelva la ecuación

$$3x \equiv 2 \bmod 78.$$

2. Resuelva el sistema de ecuaciones

$$\begin{aligned} x &\equiv 7 \bmod 9 \\ x &\equiv 13 \bmod 23 \\ x &\equiv 1 \bmod 2 \end{aligned}$$

3. Resuelva el sistema de ecuaciones

$$\begin{aligned} 2x &\equiv 3 \bmod 5 \\ 4x &\equiv 3 \bmod 7 \end{aligned}$$

4. Construya una función universal para los polinomios de una sola variable con coeficientes en Z, i.e. para la clase $Z[x]$.

5. Muestre que toda familia finita de funciones posee una función universal.

6. Diremos que una función $G : \mathbb{N} \to \mathbb{N}$ es *general* si para cualquier $i \geq 0$ existe una función computable total $g_i : \mathbb{N} \to \mathbb{N}$ tal que

$$\forall x : \ G(g_i(x)) = f_i^{(1)}(x).$$

a) Muestre que existen funciones generales.

Sugerencia: Considere una función universal $U^{(1)}$, la enumeración de Cantor c y sus respectivas proyecciones *car* y *cdr*. Considere $G = U \circ (car, cdr)$.

b) Muestre que existen funciones que no son generales.

c) Muestre que existe una infinidad de funciones generales.

7. Sea $c : \mathbb{N}^2 \to \mathbb{N}$ la enumeración de Cantor y sea $c^k : \mathbb{N}^k \to \mathbb{N}$ su k-ésima iteración:

$$c^k(x, \mathbf{x}^{(k-1)}) = c(x, c^{k-1}(\mathbf{x}^{(k-1)})).$$

a) Bosqueje la construcción de un programa-**while** que calcule c. Sea m el número de variables en este programa.

b) Bosqueje la construcción de un programa-**while** que calcule c^k utilizando a lo sumo $m + k$ variables.

8. Muestre que la clase de funciones recursivas $\mathbb{N} \to \mathbb{N}$ contiene a su propia función universal.

9. Sea $\mathcal{F}$ la mínima clase de funciones $\mathbb{N} \to \mathbb{N}$ que contiene a la función *cero* y a la *diagonal* $x \mapsto a(x, x)$, donde a es una función (computable) de apareamiento, y que es cerrada bajo el esquema de composición de funciones.

Muestre que $\mathcal{F}$ no puede contener a ninguna función universal suya.

Sugerencia: Revise el Lema de la Diagonal.

10. Muestre que $Z[x]$ no puede contener a ninguna función universal suya.

11. Sea $F : \mathbb{N} \to \mathbb{N}$ la función

$$x \mapsto F(x) = \begin{cases} 1 & \text{si } f_x^{(1)}(0) \downarrow, \\ 0 & \text{si } f_x^{(1)}(0) \uparrow. \end{cases}$$

Muestre que F no puede ser recursiva.

12. Sea $\Theta : \mathbb{N} \to \mathbb{N}$ la función

$$x \mapsto \Theta(x) = \begin{cases} \sum\limits_{i \leq x} f_i^{(1)}(x) & \text{si } \forall i \leq x : f_i^{(1)}(x) \downarrow, \\ \perp & \text{en otro caso.} \end{cases}$$

Muestre que Θ sí es recursiva.

13. Decida si acaso la función $F : \mathbb{N} \to \mathbb{N}$ tal que

$$x \mapsto F(x) = \begin{cases} 1 & \text{si } f_x^{(1)}(x) = 1, \\ 0 & \text{en otro caso.} \end{cases}$$

es recursiva. Justifique su respuesta.

14. Muestre que la función

$$F(i, j) = \begin{cases} 1 & \text{si } \exists k < j : f_i^{(1)}(k) \downarrow, \\ \perp & \text{en otro caso.} \end{cases}$$

es computable.

Sugerencia: Dado (i, j) "simule" la corrida en paralelo de los cálculos de los valores $f_i^{(1)}(0), \ldots, f_i^{(1)}(j-1)$, parándose con el valor requerido cuando se pare el primero, en pararse, de esos cálculos.

15. Muestre que la función

$$F(i, j) = \begin{cases} 1 & \text{si } \exists k > j : f_i^{(1)}(k) \downarrow, \\ \perp & \text{en otro caso.} \end{cases}$$

es computable.

Sugerencia: Proceda como en el ejercicio anterior recorriendo, digamos que mediante la enumeración de Cantor, a los elementos

$$s_{kl}^{ij} = l\text{-ésimo estado al que llega el cálculo de } f_i^{(1)}(k),$$

con $k > j$, $l \geq 0$.

16. Sean $g, h : \mathbb{N} \to \mathbb{N}$ dos funciones computables. Construya una función computable $f : \mathbb{N} \to \mathbb{N}$ que satisfaga las siguientes dos condiciones

$$\begin{array}{ll} \text{i)} & \text{dom}(f) = \text{dom}(g) \cup \text{dom}(h) \\ \text{ii)} & \forall x \in \text{dom}(f) : f(x) = g(x) \vee f(x) = h(x) \end{array}$$

17. Muestre que la función

$$F(i) = \begin{cases} 1 & \text{si } f_i^{(1)} \not\equiv \mathit{cero}, \\ \perp & \text{en otro caso.} \end{cases}$$

es computable.

18. Definiremos una nueva enumeración

$$\eta : \mathbb{N} \to \{\text{funciones recursivas}\}$$

de las funciones recursivas $\mathbb{N} \to \mathbb{N}$ como sigue:

Sea $a : \mathbb{N}^2 \to \mathbb{N}$ una función computable de apareamiento con proyecciones respectivas car^a, cdr^a.

Dado $m \in \mathbb{N}$, escribamos $m_1 = car^a(m)$ y $m_2 = cdr^a(m)$. Entonces $g_m = \eta(m)$ es la función definida por las siguientes dos relaciones:

$$\begin{array}{rcl} g_m(0) & = & \begin{cases} m_1 - 1 & \text{si } m_1 \neq 0, \\ \perp & \text{en otro caso.} \end{cases} \\ \forall x \neq 0 : \; g_m(x) & = & f_{m_2}^{(1)}(x) \end{array}$$

Muestre que la enumeración anterior es biyectiva, es decir

$$\begin{array}{rcl} \forall m, n & : & \eta(m) \equiv \eta(n) \;\Rightarrow\; m = n \\ \forall i \exists m & : & f_i^{(1)} \equiv \eta(m) \end{array}$$

19. Considere la enumeración η del ejercicio anterior. Muestre que existe una función recursiva H tal que

$$\forall m : \quad \eta(m) \equiv f_{H(m)}.$$

20. Decida si acaso existe una función recursiva G "inversa" de la anterior, es decir, tal que

$$\forall i : \quad f_i \equiv \eta(G(i)).$$

12.6 Tercera lista de programas

1. *Teorema de Cook*: Implemente la demostración del teorema de Cook. Dada una máquina de Turing y una descripción inicial construya la instancia de SAT que es equivalente a la convergencia de la máquina con la descripción dada.

   Ver: Hopcroft, Ullman: "Int. to automata theory, languages and computation". pp. 325-327.

2. Convierta instancias de SAT a instancias equivalentes de CNF-SAT. Convierta soluciones.

   Ver: Hopcroft, Ullman: "Int. to automata theory, languages and computation". pp. 328-330.

3. Convierta instancias de SAT a instancias equivalentes del problema de la programación entera. Convierta soluciones.

Ver: Hopcroft, Ullman: "Int. to automata theory, languages and computation". pp. 338.

4. Convierta instancias de 3CNF a instancias equivalentes de "Vertex_Cover". Convierta soluciones.

Ver: Hopcroft, Ullman: "Int. to automata theory, languages and computation". pp. 331-332.

5. Convierta instancias de 3CNF a instancias equivalentes de "circuito hamiltoniano dirigido". Convierta soluciones.

Ver: Hopcroft, Ullman: "Int. to automata theory, languages and computation". pp. 333-335.

6. Convierta instancias de "circuito hamiltoniano dirigido" a instancias equivalentes de "circuito hamiltoniano no-dirigido". Convierta soluciones.

Ver: Hopcroft, Ullman: "Int. to automata theory, languages and computation". pp. 335.

7. Convierta instancias de 3CNF a instancias equivalentes de 3DM. Convierta soluciones.

Ver: Even: "Graph algorithms", Computer Science Press, Maryland. pp. 205-207.

8. Convierta instancias de 3DM a instancias equivalentes de 3XC. Convierta soluciones.

Ver: Even: "Graph algorithms", Computer Science Press, Maryland. pp. 207.

9. Convierta instancias de 3CNF a instancias equivalentes de 3C. Convierta soluciones.

Ver: Even: "Graph algorithms", Computer Science Press, Maryland. pp. 218-219.

10. Convierta instancias de 3C a instancias equivalentes de 3CP. Convierta soluciones.

Ver: Even: "Graph algorithms", Computer Science Press, Maryland. pp. 221-223.

11. Convierta instancias de 3CNF a instancias equivalentes de CC. Convierta soluciones.

Ver notas del curso.

12. Convierta instancias de 3CNF a instancias equivalentes de DSP. Convierta soluciones.

Ver notas del curso.

13. Convierta instancias de 3CNF a instancias equivalentes del problema de asignamiento de recursos a tareas. Convierta soluciones.

Ver notas del curso.

Bibliography

[1] Aho, Ullman: *Foundations of computer science*, W. H. Freeman & Co., 1992

[2] Aho, Hopcroft, Ullman: *The design and analysis of computer algorithms*, Addison-Wesley, Reading, MA., 1974

[3] Anderson, Turing *et al*: *Mentes y máquinas*, en la colección *Problemas científicos y filosóficos*, Universidad Nacional Autónoma de México, 1970

[4] Arbib, Kfoury, Moll: *A basis for theoretical computer science*, Springer-Verlag, 1981

[5] Balcázar. Díaz, Gabarró: *Structural complexity I*, Springer-Verlag, 1988

[6] Barwise: *Handbook of mathematical logic*, North-Holland, 1977

[7] Chaitin: *Algorithmic information theory*, Cambridge University Press, 1987

[8] Cutland: *Computability: An introduction to recursive function theory*, Cambridge University Press, 1980

[9] Davis: *Computability and unsolvability*, Mc-Graw Hill, 1958

[10] Davis: *The undecidable*, Raven Press, 1965

[11] Epstein: *Degrees of unsolvability: Structure and theory*, Lect. Notes in Comp. Sci. Nr. 759, Springer-Verlag, 1979

[12] Garey, Johnson: *Computers and intractability: A guide to the theory of NP-completeness*, Freeman, 1979

[13] Grzegorczyk: *Some classes of recursive functions*, Rosprawy matematyczne Nr. 4, IMPAN, 1953

[14] Hermes: *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit*, Springer-Verlag, 1965 (English translation: *Enumerability, decidability and computability*, Springer-Verlag, 1969)

[15] Hinman: *Recursion-theoretic hierarchies*, Springer-Verlag, 1978

[16] Hofstadter: *Gödel, Escher, Bach: An eternal golden braid*, Vintage Books, 1980

[17] Hopcroft, Ullman: *Introduction to automata theory, languages and computation*, Addison-Wesley, 1979

[18] Johnson: A catalog of complexity classes, in [35].

[19] Johnson: The NP-completeness column: an ongoing guide, serie de artículos publicados periódicamente en *Journal of Algorithms*, de 1981 a 1988.

[20] Kalmár: An argument against the plaussibility of Church's thesis, in Heyting (ed.) *Constructivity in matematics: Proceedings of the colloquium held in Amsterdam, 1957*, North-Holland, 1959

[21] Kfoury, Moll, Arbib: *A programming approach to computability*, Springer-Verlag, 1982

[22] Kleene: *Introducción a la metamatemática*, Col. Estructura y Función, Ed. Tecnos, Madrid, 1974

[23] Lewis, Papadimitiou: *Elements of the theory of computation*, Prentice Hall, 1981

[24] Malc'ev: *Algorithms and recursive functions*, Wolters-Noordhoff Publishing Co, 1970

[25] Machtey, Young: *An introduction to the general theory of algorithms*, North-Holland, 1978

[26] Papadimitiou, Steiglitz: *Combinatorial optimization: Algorithms and complexity*, Prentice-Hall, 1982

[27] Penrose: *The emperor's new mind: Concerning computers, minds and the laws of physics*, Oxford University Press, 1989

[28] Rogers: *Theory of recursive functions and effective computability*, McGraw-Hill, 1967

[29] Rosenblueth: *Mente y cerebro: Una filosofía de la ciencia*, Siglo XXI, 1970

[30] Savage: *The complexity of computing*, Wiley, 1976

[31] Salomaa: *Formal languages*, Academic Press, 1973

[32] Shanon, McCarthy (ed's): *Automata studies*, Princeton University Press, 1956

[33] Shoenfield: *Degrees of unsolvability*, North-Holland, 1971

[34] Soare: *Recursively enumerable sets and degrees of unsolvability*, Springer-Verlag, 1987

[35] van Leeuwen (ed.): *Handbook of theoretical computer science*, North-Holland, 1990

[36] Yasuhara: *Recursive function theory and logic*, Academic Press, 1990