Breves Notas sobre
Teoría de la Computación

Jorge L. Ortega Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM

Enero 2008

Índice general

1. El Teorema de Gödel
Límites en Lógica **7**

2. La Máquina de Acceso Aleatorio
Una Computadora Abstracta **15**

3. No-determinismo
Un Autómata que Supone Correctamente **23**

4. Máquinas de Turing
Las Computadoras Más Sencillas **31**

5. Máquinas Universales de Turing
Computadoras como Programas **41**

6. Cálculo de Predicados
El Método Resolución **47**

7. El Problema de la Detención
Lo No-Computable **57**

8. El Problema de la Palabra
Diccionarios como Programas **63**

Prefacio

Las *Breves Notas sobre Teoría de la Computación* presentan en forma simple y sencilla algunos temas relevantes de Teoría de la Computación. No tienen la intención de substituir a los diversos libros y publicaciones formales en el área, ni cubrir por completo los cursos relacionados, sino más bien, su objetivo es exponer brevemente y guiar al estudiante a través de los temas que, por su relevancia, se consideran esenciales para el conocimiento básico de esta área, desde una perspectiva del estudio de la Computación.

Los temas principales que se incluyen en estas notas son: El Teorema de Gödel, la Máquina de Acceso Aleatorio, No-determinismo, Máquinas de Turing, Máquinas Universales de Turing, Cálculo de Predicados, el Problema de la Detención y el Problema de la Palabra. Estos temas se exponen haciendo énfasis en los elementos que el estudiante (particularmente el estudiante de Computación) debe comprender en las asignaturas que se imparten como parte de la Licenciatura en Ciencias de la Computación, Facultad de Ciencias, UNAM.

Jorge L. Ortega Arjona
Enero 2008

Capítulo 1

El Teorema de Gödel
Límites en Lógica

A principios de los años 1930s, Kurt Gödel, un matemático alemán, intenta probar que el cálculo de predicados era "completo", es decir, que se puede obtener mecánicamente (en principio, al menos) una demostración de cualquier fórmula verdadera expresada en tal cálculo. Su falla en lograr esto se corona con el descubrimiento de que tal tarea es imposible: ciertos sistemas formales, incluyendo la aritmética, son incompletos en este sentido. Este descubrimiento impactó al mundo de las matemáticas.

Como parte del Programa de las Matemáticas para el Cambio de Siglo de David Hilbert, se esperaba que todas las matemáticas, cuando se encontraran adecuadamente formalizadas en un sistema como el cálculo de predicados, resultaran ser completas. Pero Gödel descubre que ni siquiera la aritmética es completa. Su ahora famoso teorema establece que en cualquier sistema sólido, consistente y formal que contiene a la artimética, hay declaraciones que no pueden ser demostradas; declaraciones cuya veracidad se conoce por otros medios, pero no por ningún proceso de decisión formal o paso a paso.

Desde el tiempo de Hilbert, una sucesión de investigadores han intentado formular tal proceso de decisión en varias formas. Todos tenían en común, sin embargo, la adopción de ciertos axiomas y una variedad de formalismos para manipular los axiomas mediante reglas para obtener nuevas declaraciones que, dada la veracidad de los axiomas, resultaban ciertas. Uno de tales sistemas es la teoría de las funciones recursivas, que el propio Gödel participó en su desarrollo. Las funciones recursivas son una más de las descripciones de lo que significa computar.

El punto central del método de Gödel es codificar cada declaración posible en el cálculo de predicados, visto como un lenguaje, mediante códigos numéricos especiales conocidos como “números de Gödel”. Brevemente, el proceso tiene tres pasos:

- Proponer axiomas para el cálculo de predicados junto con una regla de inferencia mediante la cual se pueden obtener nuevas fórmulas de fórmulas iniciales.
- Proponer axiomas para la aritmética estándar en el lenguaje del cálculo de predicados.
- Definir una numeración para cada fórmula o secuencia de fórmulas en el sistema formal resultante.

Utilizando un lenguaje implicativo fácil de leer, los axiomas para el cálculo de predicados pueden listarse como sigue:

1. $\forall y_i(F \to (G \to F))$
2. $\forall y_i((F \to (G \to H)) \to ((F \to G) \to (F \to H)))$
3. $\forall y_i((\neg F \to \neg G) \to ((\neg F \to G) \to F))$
4. $\forall y_i(\forall x(F \to G) \to (F \to \forall x G))$ siempre que F no tenga ocurrencia independiente de x
5. $\forall y_i((F \to G) \to (\forall y_i F \to \forall y_i G))$
6. $\forall y_i(\forall x F(x) \to F(y))$ siempre que y no se cuantifique cuando se substituye

Una vez entendida la notación anterior, los axiomas parecen razonablemente auto-evientes como la base de una teoría de deducción matemática.

Los símbolos como F, G y H se entiende que representan “fórmulas”; $\forall y_i$ representa una cadena arbitraria de variables como $y_1, y_2, \ldots, y_k$, todas universalmente cuantificadas; tal simbolismo de lee “para todas y_i” ó “para todas $y_1, y_2, \ldots, y_k$”. El símbolo $\to$ representa a la implicación, y $\neg$ a la negación.

Las varias fórmulas involucradas en los axiomas anteriores pueden o no contener a las variables que se cuantifican fuera de ellas, pero en cualquier

caso, el axioma 4 no puede aplicarse a menos que cada vez que ocurra la variable x en la fórmula F, debe estar cuantificada dentro de F. El axioma 6 no puede aplicarse a menos que y no esté cuantificada dentro de $F(y)$ (esto es, la fórmula que se obtiene al reemplazar todas las ocurrencias independientes de x en F por y).

Los axiomas por sí mismos pueden ser fácilmente entendibles. Por ejemplo, el axioma 1 puede leerse como "para todos los valores posibles de sus variables independientes, si F es verdadero, entonces $G \rightarrow F$". En otras palabras, una fórmula verdadera se implica por *cualquier* fórmula. El axioma 2 dice que la implicación es distributiva sobre sí misma. Y el axioma 3 sostiene que si ambos $\neg G$ y G se implican por $\neg F$, entonces $\neg F$ no puede ser verdadera, es decir, F debe ser verdadera. Como un ejemplo final, el axioma 4 se puede leer como "para todos los valores posibles de sus variables independientes (como siempre), si $F \rightarrow G$ para toda x y si x no tiene ocurrencias independientes en F, entonces $F \rightarrow \forall x G$".

Al conjunto anterior de axiomas se debe añadir una regla de inferencia como la siguiente:

$$\text{Si } F \text{ y } F \rightarrow G, \text{ entonces } G$$

Es notorio que esta regla no cae en el mismo nivel que los axiomas, en un sentido: se pretende que cada vez que se haga una deducción (esencialmente, una cadena de fórmulas) y se observen fórmulas F y $F \rightarrow G$ ambas ocurriendo como miembros anteriores de la cadena, se puede añadir G a la cadena.

Se entiende como "aritmética estándar" simplemente a los Postulados de Peano para los números naturales:

1. $\forall x \, \neg(0 = sx)$
2. $\forall x, y \, (sx = sy) \rightarrow (x = y)$
3. $\forall x \, x + 0 = x$
4. $\forall x, y \, x + sy = s(x + y)$
5. $\forall x, y \, x \times sy = xxy + x$
6. $\forall x \, x \times 0 = 0$

Aquí, s denota la función sucesor, la cual para cada número natural x, arroja a su sucesor $x+1$. Así, los postulados 1 y 2 afirman, respectivamente, que:

- Cero no es el sucesor de ningún número natural.
- Si los sucesores de dos números son iguales, entonces esos números son iguales.

El resto de los postulados son relativamente fáciles de entender. Sin embargo, este primer conjunto de seis postulados requieren complementarse con el concepto de igualdad. Esto se encuentra plasmado en los siguientes tres postulados:

1. $\forall x \, x = x$
2. $\forall x, y, z \, (x = y) \rightarrow ((x = z) \rightarrow (y = z))$
3. $\forall x, y \, (x = y) \rightarrow (A(x, x) \rightarrow A(x, y))$

donde A es cualquier fórmula que tiene dos variables independientes.

Así como se ha añadido una regla especial de inferencia a los axiomas para el cálculo de predicados, se añade ahora una regla de inducción:

$$(P(0) \& \forall x (P(x) \rightarrow P(sx))) \rightarrow \forall x P(x)$$

Esta fórmula codifica la bien conocida regla de inducción: si un predicado P es verdadero al substituirse en él el valor 0 y si en cualquier caso P es verdadero para un número x, P es también verdadero para el sucesor de x, entonces P es verdadero para todos los números posibles x.

Los 15 axiomas y dos reglas que se han listado hasta aquí son lo suficientemente poderosos para proporcionar un sistema formal para la aritmética, en el cual muchas ideas pueden expresarse y verdades comprobarse, que es tentador imaginar *a priori* que cualquier verdad aritmética es no sólo expresable en este sistema, sino también demostrable en el mismo.

Habiendo preparado estos axiomas, Gödel continúa con el siguiente paso de su demostración, es decir, asigna un número único a cada fórmula concebible en el sistema que se ha definido. Hace esto mediante asignar un número natural a cada uno de los siguientes símbolos básicos:

Símbolo	Código
0	1
s	2
$+$	3
$\times$	4
$=$	5
(	6
)	7
,	8
x	9
1	10
$\neg$	11
&	12
	13
$\forall$	14
$\rightarrow$	15

Ahora bien, dado cualquier axioma o fórmula dentro del sistema formal anterior, es cosa fácil revisar la fórmula de izquierda a derecha, reemplazando cada símbolo en ella por un número primo elevado a la potencia del código del símbolo. Los primos utilizados para esto son consecutivamente 2, 3, 5, 7, 11, etc. Como ejemplo, considérese el axioma 4 anterior:

$$x_1 + sx_{11} = s(x_1 + x_{11})$$

que tiene el número de Gödel:

$$2^9 \cdot 3^{10} \cdot 5^3 \cdot 7^2 \cdot 11^9 \cdot 13^{10} \cdot 17^{10} \cdot 19^5 \cdot 23^2 \cdot 29^6 \cdot 31^9 \cdot 37^{10} \cdot 41^3 \cdot 43^9 \cdot 47^{10} \cdot 53^{10} \cdot 59^7$$

El número se obtiene de revisar la expresión dada símbolo a símbolo, convirtiéndolo en la potencia prima apropiada. Así, x, el primer símbolo, tiene código 9; por lo tanto, 2 (el primer primo) se eleva a la novena potencia. El siguiente símbolo, 1, se convierte en 3^{10}, ya que 3 es el siguiente primo y 10 es su código.

Nótese que el axioma ha sido alterado de su forma original por comodidad para usar las variables de notación especial propuestas, por ejemplo, el uso de notación unaria (consistente en unos consecutivos) como subíndices del símbolo x. Es decir, x_1 y x_{11} son considerados nombres perfectamente generales como x y y en el axioma 4 de la aritmética.

Como puede verse en el ejemplo anterior, los números de Gödel tienden a ser enormes. Sin embargo, son computables, y dado cualquier entero, es posible computar la expresión que representa (si la hay) mediante encontrar todos sus factores primos y agruparlos como potencias en el orden incremental de primos.

En este momento, se puede acceder al centro del Teorema de Gödel mediante considerar el siguiente predicado:

$$Proof(x, y, z)$$

Aquí, $Proof(x, y, z)$ es un predicado que tiene la siguiente interpretación: "x es el número de Gödel de una demostración X de una fórmula Y (con una variable independiente y número de Gödel y) la cual tiene al entero z subtituido dentro de sí". La "demostración X" a la que se refiere aquí puede considerarse a sí misma como una fórmula a fin de que se tenga un número de Gödel asignado.

Nótese que los símbolos básicos a los que los códigos se conectan no incluyen el símbolo "Proof" o ningún otro símbolo de predicado excepto la igualdad (=). Ciertamente, "$Proof(x, y, z)$ es tan solo la forma resumida para una expresión inmensamente larga con tres variables independientes x, y y z, o en la notación de Gödel, x_1, x_{11} y x_{111}. Esta expresión incluye un número de procedimientos, incluyendo los siguientes:

1. Dado un entero, se produce una cadena de la cual éste es su número de Gödel.

2. Dada una cadena, se verifica si es una fórmula.

3. Dada una secuencia de fórmulas, se verifica si es una demostración para la última fórmula en la secuencia.

Todos estos procedimientos son computables y, como Gödel demuestra, en sí mismos reducibles a fórmulas dentro del sistema formal definido anteriormente. Antes de mostrar cómo este predicado se usa en el Teorema de Gödel, hay un pequeño detalle que aclarar. Se debe establecer qué es una fórmula: la definición que se encuentra en el procedimiento 2 anterior aportaría a una definición inductiva de una expresión aritmética propiamente formada y las formas en las cuales tales expresiones pueden legalmente combinarse mediante conectivas lógicas & y $\rightarrow$, o cuantificadas con $\forall$ y $\exists$.

Por ejemplo, $\exists x_1(X_{11}(x) = X_{111}(x))$ es una fórmula, pero $1X)\exists x_{11}\neg =))$ no lo es.

Ahora bien, se puede proceder a considerar un uso muy especial del predicado bajo consideración. Supóngase que a la fórmula Y se le alimenta su propio número de Gödel y que se niega la existencia de una demostración dentro del sistema formal de la siguiente fórmula:

$$\neg\exists x Proof(x, y, z)$$

Es decir, la fórmula $Proof(x, y, z)$ significa "x es el número de Gödel de una demostración de la fórmula obtenida mediante substituir su propio número de Gödel y por su única variable independiente". Consecuentemente, escribir $\neg\exists x$ al principio de esto es negar la existencia de tal demostración.

Tal predicado expresable en el sistema formal que se ha construido tiene un número de Gödel. Sin embargo, considérese lo siguiente:

- En primera instancia, se tiene la expresión con una sola variable independiente $\neg\exists x Proof(x, y, z)$, que se alimenta con y. Su propio número de Gödel se representa con g.
- En seguida, a la propio expresión se le alimenta con su propio número de Gödel g, y al substituirlo, se transforma en un predicado sin variables independientes que no puede recibir más valores para sus variables.
- Naturalmente, dado lo anterior, la fórmula resultante tiene ahora un nuevo número de Gödel g'.

> *Teorema de Gödel:* $\neg\exists x Proof(x, g, g)$ es verdadero pero no demostrable en el sistema formal de la artimética.

La demostración de esto requiere tan solo unas cuantas líneas:

> Supóngase que $\neg\exists x Proof(x, g, g)$ es demostrable en el sistema, y sea p el número de Gödel de tal demostración P. Se tiene entonces que:
>
> $$Proof(p, g, g)$$
>
> es verdadero ya que P es una demostración de G con g substituido por su única variable independiente. Pero, evidentemente, $Proof(p, g, g)$ contradice $\neg\exists x Proof(x, g, g)$, arrojando la conclusión de que no existe tal demostración P.

La fórmula $\neg\exists x Proof(x, g, g)$ es ciertamente verdadera porque se ha establecido la propuesta que hace de sí misma: que no tiene demostración.

Tal demostración, donde a predicados de dos variables se les da el mismo valor para sus argumentos, se le conoce como "demostración por diagonalización", y aparece frecuentemente en la teoría de los conjuntos infinitos y en la lógica matemática. Cantor es el primero en usar tal argumento para demostrar que los números reales no son contables.

¿Hay afirmaciones verdaderas que los matemáticos intentan demostrar ahora mismo, pero nunca lo harán? ¿Qué tal la Conjetura de Goldbach, que establece que todo número par es la suma de dos primos? Ciertamente, nadie ha demostrado esta afirmación hasta hoy, aun cuando muchos matemáticos creen que es verdadera.

La lucha por formalizar las matemáticas en una forma mecánica llevó al descubrimiento de un problema básico y profundamente enraizado en las propias matemáticas. El descubrimiento se considera a la altura del intento unos años más tarde de formalizar los "procedimientos efectivos", lo que derivó en el descubrimiento de una inadecuación básica de las computadoras. Hay algunas tareas que simplemente son imposibles tanto para las computadoras como para los matemáticos.

Capítulo 2

La Máquina de Acceso Aleatorio

Una Computadora Abstracta

El término “máquina de acceso aleatorio” (*random access machine* ó RAM) tiende a tener un doble uso dentro de la computación. A veces, se refiere a computadoras específicas con memorias de acceso aleatorio, es decir, memorias en acceso a una dirección es tan fácil y sencillo como el acceso a cualquier otra dirección. Por otro lado, se refiere a un cierto modelo de cómputo que no es menos “abstracto” que, por ejemplo, la máquina de Turing, pero que está más cercana en cuanto a su operación a las computadoras digitales programables estándar.

No es de sorprender que la principal diferencia entre las máquinas de Turing y las RAM (abstractas) recae en el tipo de memoria empleado. Por un lado, la memoria de una máquina de Turing está completamente contenida en una cinta, haciéndola una especie de “máquina de acceso serial”. Una RAM, por otro lado, tiene su memoria organizada en palabras, y cada palabra tiene una dirección. Tiene, más aún, un número de registros. El modelo descrito aquí tiene un registro llamado “acumulador” (*accumulator*, ó AC).

Como es el caso con las máquinas de Turing, las RAM se definen en términos de programas, el tipo de estructuras como la que se muestra en la figura 2.1, siendo meramente un vehículo para interpretar tales programas. Como tal, la figura muestra a la RAM equipada con una Unidad de Control capaz de realizar las operaciones especificadas en el Programa de la RAM, el cual está “cargado” en algún sentido. El Programa dicta la transferencia

de información entre varias palabras de memoria y el acumulador. También especifica ciertas operaciones sobre los contenidos del acumulador. Finalmente, es capaz de dirigir cuál de sus propias instrucciones debe ejecutarse enseguida.

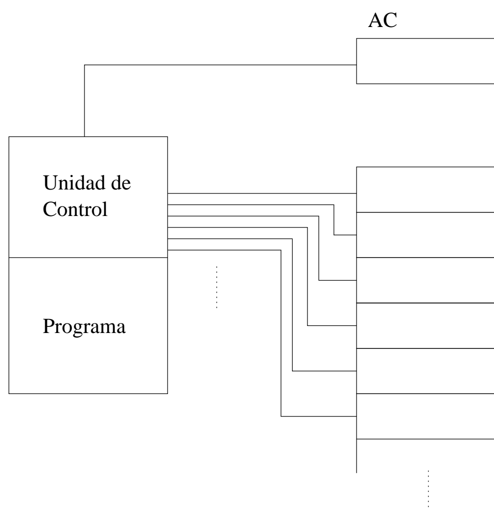


Figura 2.1: Una máquina de acceso aleatorio.

Tanto el acumulador como cada una de las palabras de memoria se suponen capaces de contener un solo entero, sin importar su longitud. Aun cuando la RAM tiene solo un registro, tiene un número infinito de palabras en su memoria. En la figura se muestra una línea de comunicación entre la Unidad de Control y cada una de sus palabras de memoria. Esto simboliza el hecho de que tan pronto como una dirección se especifica en el Programa de la RAM, la palabra de memoria correspondiente a esa dirección se hace instantáneamente accesible a la Unidad de Control.

Se conoce una gran variedad de RAM, definidas por varios autores. Sin embargo, aquí sólo se muestra una clase muy sencilla, que es programable en el lenguaje que se resume en la siguiente tabla. A este lenguaje, se le conoce con el nombre de "lenguaje de acceso aleatorio" (*random access language*, ó RAL).

Nemónico	Argumento	Significado
LDA	X	Carga en AC con el contenido de la dirección de memoria X
LDI	X	Carga en AC indirectamente con el contenido de la dirección de memoria X
STA	X	Almacena el contenido de AC en la dirección de memoria X
STI	X	Almacena el contenido de AC indirectamente en la dirección de memoria X
ADD	X	Suma el contenido de la dirección de memoria X con el contenido de AC
SUB	X	Resta el contenido de la dirección de memoria X del contenido de AC
JMP	X	Salta a la instrucción con etiqueta X
JMZ	X	Salta a la instrucción con etiqueta X si AC contiene 0
HLT		Alto

Cuando se habla del contenido de la dirección de memoria X, se refiere al entero contenido actualmente en la palabra de memoria cuya dirección es X. Para cargar AC *indirectamente* con los contenidos de la dirección de memoria X significa tratar a los contenidos de la palabra de memoria como otra dirección cuyos contenidos serán cargados. La instrucción STI utiliza la misma clase de indirección, pero en sentido inverso.

Cuando se escribe un programa RAL, normalmente se numeran sus instrucciones a fin de que las instrucciones de salto (JMP y JMZ) pueden comprenderse claramente. Así, JMP 5 significa que la siguiente instrucción a ser ejecutada es la instrucción en la dirección 5. JMZ 5 significa ejecutar la instrucción con dirección 5 siempre y cuando el contenido de AC sea 0; de otra manera, continúa con la siguiente instrucción despues de JMZ.

El indireccionamiento es una característica muy útil en los programas RAM. Ambas instrucciones, LDI y STI, utilizan la indirección de la siguiente manera: LDI X significa primero buscar el contenido de X. Si el contenido almacenado aquí es Y, entonces la RAM enseguida busca los contenidos de Y, finalmente cargando el contenido de esa palabra de memoria a AC. Esta forma de referencia indirecta de memoria también ocurre para STI: busca los contenidos de X, obtiene el entero almacenado ahí, por ejemplo Y, y entonces almacena el contenido de AC en la palabra de memoria Y.

Como en el caso de las máquinas de Turing, no se toma en cuenta la entrada o salida de la RAM; ya que no se trata de máquinas “reales”, no hay razón en tratar de comunicarlas con el usuario. En lugar de esto, un cierto conjunto inicial y finito de enteros se supone que existe en ciertas palabras

de memoria, con el resto conteniendo ceros. Al final de un procesamiento de la RAM, lo que permanece en memoria se considera la salida.

Una RAM se detiene bajo dos condiciones:

- Si durante la ejecución se encuentra una instrucción HLT, todas las operaciones cesan y el cómputo llega a su fin.
- Si la RAM encuentra una operación no ejecutable, entonces se detiene. Una operación no ejecutable es aquélla cuyos argumentos no hacen sentido en el contexto de la RAM. Por ejemplo, si una dirección tiene un valor negativo, de modo que STA -8 no tiene sentido. De manera similar, si se tiene JMP 24 en un programa con 20 instrucciones. Tales operaciones no ejecutables se supone que no existen en programas RAL.

A continuación, considérese un algoritmo que detecta enteros duplicados en una secuencia A de enteros positivos. Lo enteros se encuentran originalmente almacenados en otro arreglo B, cada uno igual al entero almacenado.

STOR

1. **for** $i \leftarrow 1$ **to** n **do**
   a) **if** $B(A(i)) \neq 0$
      1) **then** output $A(i)$; exit
      2) **else** $B(A(i)) \leftarrow 1$

Este algoritmo puede traducirse en un programa en RAL como sigue:

$STOR - RAL$

01. `LDI 3` / obtiene la i-ésima entrada de A
02. `ADD 4` / suma un desplazamiento para obtener el índice j
03. `STA 5` / almacena el índice j
04. `LDI 5` / obtiene la j-ésimo entrada de B
05. `JMZ 9` / si la entrada es 0, salta a 9
06. `LDA 3` / si la entrada es 1, obtiene el índice i
07. `STA 2` / y lo almacena en 2
08. `HLT` / detiene la ejecución
09. `LDA 1` / obtiene la constante 1

10. `STI 5` / y la almacena en B
11. `LDA 3` / obtiene el índice i
12. `SUB 4` / substrae el límite
13. `JMZ 8` / si $i =$ límite, se detiene
14. `LDA 3` / obtinen el índice (de nuevo)
15. `ADD 1` / incrementa i
16. `STA 3` / almacena el nuevo valor de i
17. `JMP 1` / regresa a la primera instrucción

El programa $STOR$ se puede comprender mejor si se refiere a la figura 2.2, en la cual el uso del programa tanto del AC y la memoria se muestra claramente.

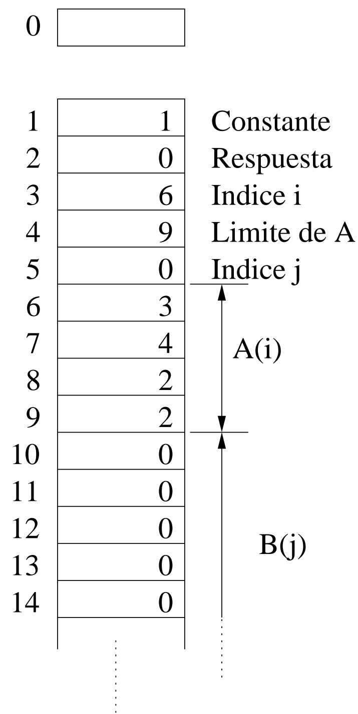


Figura 2.2: El programa $STOR$ en memoria.

Las primeras cinco palabras de la memoria contienen las variables y constantes utilizadas dentro del programa. En este caso particular, las siguientes cuatro palabras contienen un conjunto de 4 enteros que se verifica si están

duplicados. Estas palabras son colectivamente referidas como A, efectivamente, un arreglo. Todas las demás palabras de la memoria, de la dirección 10 en adelante, comprenden al arreglo infinito B, en el cual $STOR$ coloca un valor de 1 cada vez que procese un entero de A.

La primera dirección de memoria contiene la constante 1, que el programa $STOR$ utiliza para incrementar el índice i. Este índice (o más bien, su valor actual) se almacena en la dirección 3. Un segundo índice j se almacena en la dirección 5, y sirve para accesar a los contenidos en B.

Las tres primera instrucciones de $STOR$ son:

01. `LDI 3`
02. `ADD 4`
03. `STA 5`

Estas instrucciones leen el contenido de la dirección i, suma su contenido al contenido de la dirección 4, y entonces almacenan el resultado en la dirección 5. Específicamente, la instrucción LDI lee el contenido de la dirección 3, y coloca este número en AC. La dirección 4 contiene la constante 9, que delimita la sección de memoria de A. El efecto de sumar 9 con 3 en AC es obtener la tercera dirección en B, es decir, 12. La computadora entonces almacena este número en la dirección 5. Así, $STOR$ lee 3 como el primer entero de A y obtiene la dirección del tercer elemento de B.

La siguiente instrucción de $STOR$ (LDI 5) carga el número almacenado en la tercera dirección de B en AC. Este número tiene valor 0 para la instrucción 5, de modo que la ejecución salta a la instrucción 9. Las instrucciones 9 y 10 leen la constante 1 y la almacenan en la tercera dirección de B: en realidad, sucede que ésta es 12, que es donde $STOR$ coloca el valor 1. Esto indica que $STOR$ ha encontrado un 3. Si (dada otra secuencia en A) $STOR$ encontrara un 3 más tarde, tendría un valor 1 en AC cuando llegara a la instrucción JMZ 9. En tal caso, inmediatamente cargaría el valor actual de i en la dirección 3, lo almacenaría en la dirección 2, y se detendría.

La dirección 2 contiene la respuesta cuando $STOR$ termina. Si es 0, significa que no hay duplicados entre los enteros de A. Si no, contiene la dirección en A del entero repetido.

El resto del programa, de la instrucción 11 a la 16, recupera el índice i de la dirección 3 y le resta 9 (de la dirección 4) para decidir si $STOR$ ha llegado al final de A. Si este no es el caso, incrementa el índice i y lo almacena de

nuevo en la dirección 3 antes de saltar de regreso al inicio del programa, a fin de revisar al siguiente entero en la secuencia A.

A primera vista, parece que las RAM son más poderosas que las máquinas de Turing en términos de las funciones que puede operar, pero esto no es realmente cierto. Las máquinas de Turing pueden hacer todo lo que las RAM pueden hacer. Por ahora, es interesante comprobar lo inverso. Puede parecer obvio, pero ¿las RAM realmente pueden hacer cualquier cosa que una máquina de Turing puede hacer? Afortunadamente, esto no es difícil de demostrar. Dada una máquina de Turing arbitraria con una cinta semi-infinita, utilícese la memoria de una RAM para simular la cinta, como se muestra en la figura 2.3.

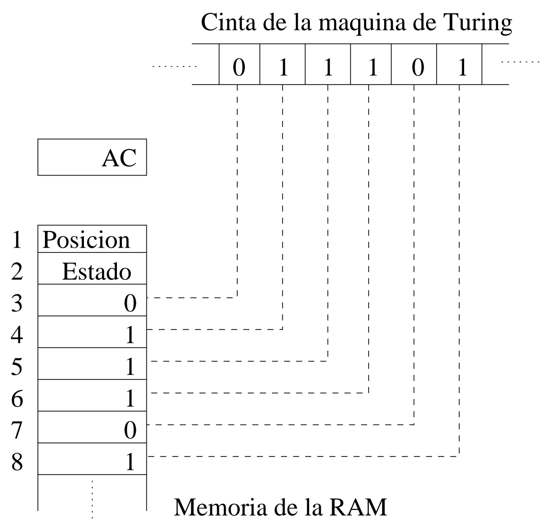


Figura 2.3: Una RAM simulando una máquina de Turing.

Dada la correspondencia uno a uno entre las palabras de la memoria de la RAM y las celdas de la cinta en la máquina de Turing, se requiere ahora tan solo escribir un programa para la RAM que simule el programa de la máquina de Turing. Cada quíntupla de la máquina de Turing, entonces, debe remplazarse por un número de instrucciones en RAL, que tengan el mismo efecto sobre la memoria de la RAM como la quíntupla lo tiene sobre la cinta de la máquina de Turing. Para tal fin, es necesario que el programa RAM "recuerde" la posición de la cabeza lectora/escritora de la máquina de Turing, así como su estado actual.

Debido a su equivalencia con máquinas de Turing y, ciertamente, por su equivalencia con todas los más poderosos esquemas computacionales abstractos, las RAM ofrecen una alternativa en el estudio de cómputos factibles. Algunas preguntas pueden formularse de manera más sencilla, y responderse de igual manera en el marco de tal modelo de cómputo; de todos los esquemas, es el más parecido a una computadora digital estándar. Es también uno de los esquemas más convenientes para expresar cómputo real, tal y como lo muestra el programa RAL que se ha examinado aquí.

Capítulo 3

No-determinismo

Un Autómata que Supone Correctamente

Supóngase la siguiente situación: es de noche, y se encuentra conduciendo por una ciudad extraña. Debe llegar a la casa de X en 10 minutos, pero no tiene la dirección. No puede telefonear a X, pues no conoce su apellido. Afortunadamente, su automóvil se encuentra equipado con un autómata no-determinístico en el panel. Tocando un botón, aparecen cinco letras en su pantalla:

L F R S B

Aquí, L significa izquierda, F significa de frente, R significa derecha, S es deténgase y B significa hacia atrás. Cada vez que llegue a una intersección, se presiona el botón y se sigue la dirección dada. Eventualmente, se llega a una intersección para la cual aparece B: ya ha pasado la casa de X. Ahora debe dar la vuelta, y presionar el botón conforme pasa cada casa. Eventualmente, aparece una S. Puede ahora tocar a la casa, donde aparece su amigo que le espera.

Es evidente que esta historia es ficción, y que no existe en realidad un autómata así. Pero son muy útiles en teoría. No sólo resultan en una cierta simplificación de la teoría de autómatas, sino también proponen algunas de las más profundas preguntas abiertas en computación.

Cada uno de los autómatas en la jerarquía de Chomsky tiene una versión determinística y no-determinística. Aquí, se estudia el modelo no-determinísti-

co de los cuatro modelos de autómatas. El primero es el autómata finito. En el diagrama de transición de estados (figura 3.1) es posible descubrir una multitud de rutas a partir del estado inicial I y hasta el estado final o de aceptación F. Sin embargo, si se pregunta qué entradas colocan a este autómata en su estado final, resulta una cierta confusión: algunos estados tienen dos o más transiciones que se disparan por el mismo símbolo de entrada. Ciertamente, una entrada como 01011 puede poner a esta máquina en cualquiera de sus varios posibles estados, uno de los cuales es el estado final. La definición de un autómata finito no-deteminístico dice que tal entrada se acepta por la máquina: si el autómata *puede* aceptar una entrada dada, entonces se considera que el autómata *acepta* la entrada.

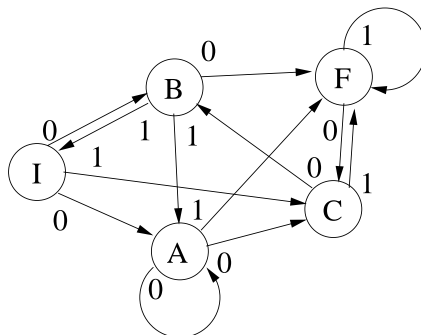


Figura 3.1: Un autómata no-determinístico.

En términos precisos, un autómata finito no-determinístico (*nondeterministic finite automaton* ó NFA) consiste de un conjunto finito de estados Q, un alfabeto finito Σ, y una función de transición δ. Lo mismo es cierto para un autómata determinístico. Sólo la función de transferencia es diferente:

$$\delta : Q \times \Sigma \rightarrow P(Q) - \varnothing$$

donde $P(Q)$ representa el conjunto de todos los subconjuntos de Q. En otras palabras, cuando el NFA está en un estado dado y un símbolo se le da como entrada, entonces resulta un conjunto no vacío de siguientes estados. En el autómata de la figura 3.1, la función de transición δ mapea el par estado-entrada $(I, 0)$ al conjunto de estados $\{A, B\}$.

Si w es una palabra compuesta de letras de Σ, se define $\delta(w)$ como el conjunto de todos los estados en los que un NFA puede encontrarse en el

momento en que el último símbolo de w le ha sido dado como entrada. Si $\delta(w)$ contiene un estado final, entonces w se acepta por el NFA. El conjunto de todas las palabras aceptadas por un NFA dado se conoce como *lenguaje*.

Es importante insistir, una vez más, que esta definición de aceptar es equivalente a decir que el autómata supone la transición correcta a cada paso del proceso. Por ejemplo, la palabra de entrada 01011 podría resultar en una serie de transiciones como $I \to B \to I \to B \to I \to C$ ó en otra serie como $I \to A \to F \to C \to F \to F$. Hasta determinar la aceptación de 01011 por el autómata, sin embargo, sólo una secuencia como la última se supone que se computa.

Podría suponerse que un autómata finito con tal extensión de poderes de suposición podría aceptar lenguajes más allá del poder de sus semejantes determinísticos. Es interesante, pero este no es el caso.

Dado un NFA, por ejemplo, M, se puede describir un autómata determinístico $\bar{M}$ que acepta exactamente el mismo lenguaje que M. He aquí como funciona.

Supóngase que M tiene un conjunto de estados Q, un alfabeto Σ y una función de transición δ. Ahora bien, sea $\bar{M}$ un autómata con un estado q_i por cada subconjunto no vacío Q_i de Q. Así, al escribir un estado de $\bar{M}$ como q_i, se está refiriendo también a un conjunto particular no vacío Q_i de Q. La razón de esta construcción algo exótica recae en una simple observación: aun si no se puede determinar en qué estado estará M después de una entrada dada, sí se puede determinar el conjunto de estados en los que M puede *posiblemente* estar.

Si M pudiera estar en cualquier estado del subconjunto Q_i en un momento dado, y si un símbolo σ se da como entrada, entonces M podría estar en cualquier estado del conjunto:

$$\bar{\delta}(Q_i, \sigma) = \bigcup_{q \in Q_i} (q, \sigma)$$

La función $\bar{\delta}$ es en realidad la función de transición para $\bar{M}$. Mediante denotar $\bar{\delta}(Q_i, \sigma)$ como Q_j, una multitud de transiciones en M se expresan como una sola transición en $\bar{M}$, como muestra la figura 3.2.

Dada esta definición para la función de transición $\bar{\delta}$ de $\bar{M}$, es claro que $\bar{M}$ es determinístico. Para cada estado actual y entrada, hay exactamente un

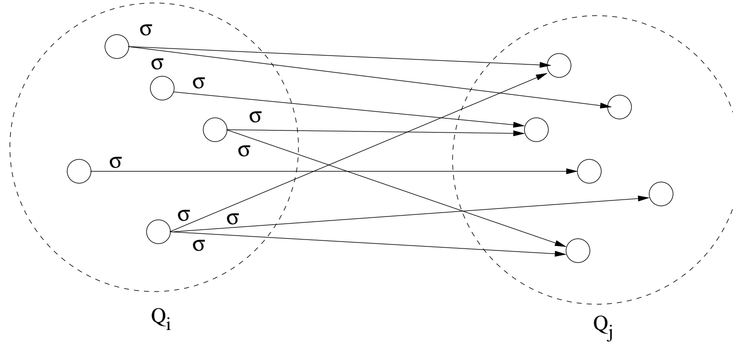


Figura 3.2: Muchas transiciones se vuelven una.

solo estado nuevo posible. Si q_0 es el estado inicial de M y F es el conjunto de los estados finales, entonces se define $Q_0 = \{q_0\}$ como el estado inicial de $\bar{M}$. Más aún, cualquier estado Q_k de $\bar{M}$ teniendo la propiedad que $Q_k \bigcap F$ no sea vacío califica como un estado final de M.

Para probar que $\bar{M}$ acepta L, el lenguaje aceptado por M, sea w una palabra arbitraria en L. Si w se compone de los símbolos

$$w_1, w_2, \ldots, w_n$$

entonces se observa que $\bar{M}$ pasa a través de una sucesión de estados

$$Q_1, Q_2, \ldots, Q_n$$

donde

$$\bar{\delta}(Q_{i-1}, w_i) = Q_i \qquad i = 1, 2, \ldots, n$$

Ahora, $Q_1, Q_2, \ldots, Q_n$ es también una sucesión de subconjuntos de Q, los estados de M. Se nota fácilmente que

$$Q_1 = \delta(q_0, w_1) \qquad Q_2 = \delta(q_0, w_1 w_2) \ldots Q_n = \delta(q_0, w_1 w_2 \ldots w_n)$$

Es decir, conforme la palabra w se da como entrada a M, puede encontrarse en cualquiera de los estados de Q_i cuando se lee el i-ésimo símbolo. Q_n contiene un estado final de M, y por lo tanto, Q_n es un estado final de M. La implicación inversa también se mantiene, si w se acepta por $\bar{M}$, entonces w también se acepta por M. Así, $\bar{M}$ acepta el mismo lenguaje que M, y el resultado queda comprobado.

Ya que todo autómata finito determinístico (*deterministic finite automaton* ó DFA) es tan solo una clase especial de un NFA, se sigue que cualquier lenguaje que se acepte por un DFA también se acepta por un NFA. Se prueba entonces que los NFA no son más podersos que los DFA cuando se trata de aceptar lenguajes. De paso se nota, sin embargo, que para obtener un DFA equivalente a un NFA dado, se requiere realizar una cantidad de trabajo exponencial para construirlo. En general, el DFA equivalente puede tener tantos como 2^n estados cuando el NFA tenga sólo n estados.

Al siguiente nivel de la jerarquía de Chomsky, se encuentra un autómata cuyas versiones determinística y no-determinística no son equivalentes en absoluto. Los autómatas de pila no-determinísticos aceptan lenguajes libre de contexto, pero algunos de estos lenguajes no se aceptan por los autómatas de pila determinísticos.

Por ejemplo, es suficientemente fácil probar que el siguiente lenguaje se acepta por algunos autómatas de pila no-determinísticos:

$$L = \{0^a 1^b 0^c 1^d : a = c \text{ ó } b = d\}$$

No resulta fácil comprobar que un autómata de pila no-determinístico puede aceptar L. Aquí, se puede argumentar intuitivamente al menos la razón de esto: al procesar la cadena de símbolos $0^a 1^b 0^c 1^d$, el autómata de pila debe de alguna manera "recordar" tanto a como b mediante almacenar información acerca de estos valores en su pila. La información puede consistir en sí misma de cadenas de ceros ó unos, o algo más sofisticado. En cualquier caso, meramente accesar la información acerca de a (para comparar a con c) resulta en una pérdida de información acerca de b. Esa información se añade a la pila *después* de la información de a, y debe sacarse (es decir, destruirse) para que el autómata pueda recordar a.

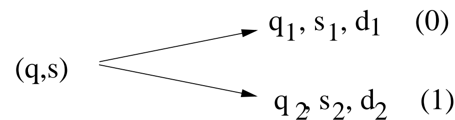
Partiendo del argumento anterior (hecho rigurosamente), el autómata de pila no-determinístico es definitivamente más poderoso que su contraparte determinística.

¿Qué sucede con las siguientes computadoras dentro de la jerarquía de Chomsky, los autómatas lineales? Estos dispositivos se parecen cada vez más a las máquinas de Turing en todos aspectos excepto uno: la cantidad de cinta que se les permite utilizar está limitada por una función lineal del tamaño de la cadena de entrada. En otras palabras, cada autómata lineal acotado tiene asociada una constante k, de modo que cuando un tamaño de palabra n aparece como entrada en la cinta, se le permite a la máquina el

uso de no más de kn celdas para determinar la aceptabilidad de la palabra. Ha sido un problema desde hace largo tiempo decidir si un autómata lineal acotado no-determinístico es más poderoso que uno determinístico. Nadie lo sabe con certeza.

En lo alto de la jerarquía de Chomsky se encuentran las máquinas de Turing. Para estos dispositivos abstractos, como para los autómatas finitos, el no-determinismo no añade nada a su capacidad de aceptar lenguajes. Dada una máquina de Turing no-determinística M, una máquina determinística equivalente M' se construye como una máquina de Turing universal.

La función de transición de M se almacena como una secuencia de quíntuplas en una cinta especial de trabajo de M'. Para simplificar el argumento, supóngase que M tiene sólo una transición no-determinística y que tal transición involucra tan solo dos alternativas, etiquetas 0 y 1:



Como en el caso de una máquina de Turing universal, M' lee la cinta de M para ver que símbolo está leyendo M. En una porción de su cinta de trabajo, M' escribe el estado actual de M'. Es entonces una cuestión simple (y sin embargo, algo complicada de describir en detalle) para M' ejecutar las quíntuplas de M para ver si alguna se aplica a la combinación de símbolo de entrada actual y estado. Si sólo hay uno, todo va bien: M' meramente escribe el símbolo a ser escrito en su versión de la cinta de M, cambia al estado escrito en la sección de estado de memoria, y mueve (simula mover) la cabeza lectora/escritora de M en la dirección apropiada.

Pero, ¿qué pasa si dos de tales transiciones son posibles? Es aquí donde M' difiere de la máquina universal. En una porción de su cinta de trabajo, mantiene una cuenta que comienza en el 0 binario y progresa al infinito si es necesario. El número almacenado en el espacio de cuenta le informa a M' cuál de las dos transacciones tomará M. Cada vez que M encuentra el par estado-símbolo (q, s) en su simulación de la actividad de M, consulta su cuenta actual. Si al consultar este número por primera vez, usa tan solo el primer dígito (0 ó 1) para decidir sobre las alternativas. Si consulta la cuenta una segunda vez, utiliza el segundo dígito, y así continúa. Tarde o temprano, acepta la palabra de entrada a M o consulta la cuenta por el

último dígito en el número. La aceptación por M' implica la aceptación por M. Pero si M' alcanza el último dígito de su cuenta sin haber aceptado la palabra de entrada a M, añade entonces un 1 a la cuenta y comienza todo de nuevo para simular M. Si M nunca se detiene, obviamente, tampoco se detiene M'. Pero la aceptación por M' es equivalente a la aceptación por M. Lo inverso de estas dos últimas afirmaciones es también cierto.

El objetivo de que M' mantenga una cuenta es que tarde o temprano los dígitos de este incremento lento y permanente del número debe coincidir con una secuencia aceptable de alternativas para las dos opciones (q, s) de M.

A lo largo de la discusión, se ha permitido sólo que el no-determinísmo resida en la selección de transiciones que el autómata pueda tomar. En el caso de las máquina de Turing, hay una segunda manera: mientras se deja a todas las transiciones como determinísticas, se puede suponer la aparición en la cinta de la máquina de Turing no-determinística de una palabra finita x en una posición inmediatamente adyancente a la palabra de entrada w. Si, en lugar de permitírsele seleccionar entre dos o más transiciones para una combinación dada de estado-símbolo, la máquina de Turing es dirigida para consultar la palabra x como una parte de su proceso de toma de decisiones, entonces una máquina equivalente al primer tipo puede siempre construirse. Esta noción particular de no-determinismo es importante al definir la muy importante noción de complejidad en tiempo polinomial no-determinístico.

Capítulo 4

Máquinas de Turing
Las Computadoras Más Sencillas

Las máquinas de Turing son los modelos teóricos de computación más sencillos y más ampliamente utilizados. Demasiado lentas y poco eficientes como para ser construidas en un dispositivo real, estas máquinas conceptuales, sin embargo, parecen capturar todo lo que significa el término *computación*. No sólo las máquinas de Turing ocupan el lugar más alto dentro de la jerarquía de Chomsky, sino también parecen capaces de computar cualquier función que sea computable mediante cualquier otro esquema conceptual de cómputo. Más aún, las máquinas de Turing son el más simple de todos esos esquemas, de funciones generales recursivas a máquinas de acceso aleatorio.

La máquina de Turing es sencilla: una cinta infinita compuesta por celdas discretas se revisa, una celda en cada momento, por una cabeza lectora/escritora que comunica con un mecanismo de control. Bajo éste, la máquina de Turing es capaz de leer el símbolo en la celda actual de la cinta, o reemplazarlo con otro. Similarmente, la máquina de Turing es capaz de desplazarse sobre la cinta, celda por celda, ya sea a la izquierda o a la derecha.

Como otras máquina de la jerarquía de Chomsky, las máquinas de Turing pueden ser tratadas como "aceptadores" de lenguajes. El conjunto de palabras que causan que una máquina de Turing llegue finalmente a detenerse se considera como su *lenguaje*. Sin embargo, como se indica anteriormente, las máquinas de Turing son mucho más que esto. En general, se considera a la función definida por una máquina de Turing M como:

Sea x una cadena sobre el alfabeto Σ de la cinta de la máquina M. Colocando la cabeza lectora/escritora sobre el símbolo más a la izquierda de x en la cinta, M comienza en su estado inicial, procediendo el cómputo sin interferencia. Si M eventualmente se detiene, entonces la cadena y que se encuentra en la cinta en ese momento se considera como la salida de M que corresponde a la entrada x. Ya que M no necesariamente se detiene para toda cadena de entrada x, M computa una función parcial:

$$f : \Sigma^* \rightarrow \Sigma^*$$

donde Σ^* denota el conjunto de todas las cadenas sobre Σ. Bajo este esquema, toda clase de posibilidades para Σ pueden tomarse. Por ejemplo, Σ puede usarse como base para la representación de enteros, racionales, símbolos alfabéticos, o cualquier clase de objetos que puedan computarse.

Estrictamente hablando, una máquina de Turing es esencialmente lo mismo que su *programa*. Formalmente, tal programa M es un conjunto de quíntuplas de la forma:

$$(q, s, q', s', d)$$

donde q es el estado actual de M, s es el símbolo que actualmente M lee en la cinta, q' es el siguiente estado de M, s' es el símbolo que M escribe en la cinta para remplazar a s, y d es la dirección a la cual la cabeza lectora/escritora se mueve relativamente sobre la cinta. Los estados provienen de un conjunto finito Q, los símbolos del alfabeto finito Σ y los movimientos de la cabeza de $D = \{L, R, S\}$ (derecha, izquierda y detente, respectivamente).

El programa de una máquina de Turing, dada una cadena particular a la entrada, comienza siempre en un *estado incial* $q_0 \in Q$ y puede detenerse en un número de formas. El método que se utiliza para detener un programa involucra el uso de *estados de detención*. Cuando M entra al estado de detención q', toda actividad computacional cesa. Naturalmente, q' nunca aparece como un símbolo inicial de las quíntuplas de un programa de M.

El programa P para máquina de Turing que se muestra a continuación se utiliza para multiplicar dos números unarios. Bajo esta representación, un número m se representa mediante m unos consecutivos sobre la cinta. Inicialmente, la cinta de P contiene dos números a ser multiplicados, separados por el símbolo $\times$. Finalmente, la cinta de P contiene el producto $m \times n$ en forma unaria. Los estados inicial y final se muestran en la figura 4.1.

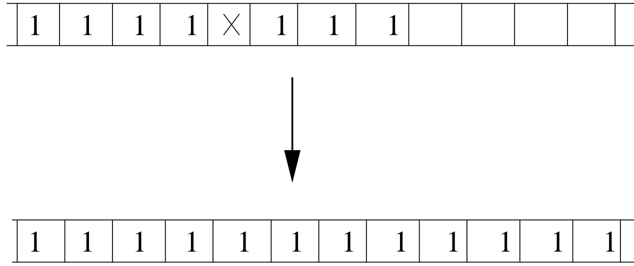


Figura 4.1: Multiplicando dos números unarios.

Como en otras máquinas conceptuales, hay muchas maneras de representar el programa de una máquina de Turing, siendo la más directa simplemente listar las quíntuplas de P.

$MULT$

$q_0, 1, q_1, 1, L$
$q_1, \flat, q_2, *, R$
$q_2, \flat, q_3, \flat, L$
$q_2, *, q_2, *, R$
$q_2, 1, q_2, 1, R$
q_2, X, q_2, X, R
q_2, A, Q_2, A, R
$q_3, 1, q_4, \flat, L$
q_3, X, q_9, X, L
$q_4, 1, q_4, 1, L$
q_4, X, q_5, X, L
$q_5, *, q_8, *, R$
$q_5, 1, q_6, A, L$
q_5, A, q_5, A, L
$q_6, \flat, q_7, 1, R$
$q_6, *, q_6, *, L$
$q_6, 1, q_6, 1, L$

$q_7, *, q_7, *, R$
$q_7, 1, q_7, 1, R$
q_7, X, q_5, X, L
q_7, A, q_7, A, R
$q_8, \flat, q_3, \flat, L$
$q_8, 1, q_8, 1, R$
q_8, X, q_8, X, R
$q_8, A, q_8, 1, R$
$q_9, *, q_{10}, \flat, S$
$q_9, 1, q_9, 1, L$
$q_{10}, \flat, q_{11}, \flat, S$
$q_{10}, 1, q_{10}, \flat, R$
$q_{10}, X, q_{10}, \flat, R$
$q_{10}, A, q_{10}, \flat, R$

La representación más comprensible del programa de una máquina de Turing es probablemente un diagrama de transición de estados. Tal diagrama puede frecuentemente ser simplificado aún más, mediante etiquetar ciertos estados como “moviéndose a la izquierda” y “moviéndose a la derecha”, y omitiendo las flechas de transición de un estado a sí mismo cuando el símbolo

que se lee se re-escribe y el movimiento de la cabeza se puede inferir con la dirección asociada a ese estado. La figura 4.2 muestra el diagrama de transición de estados para P.

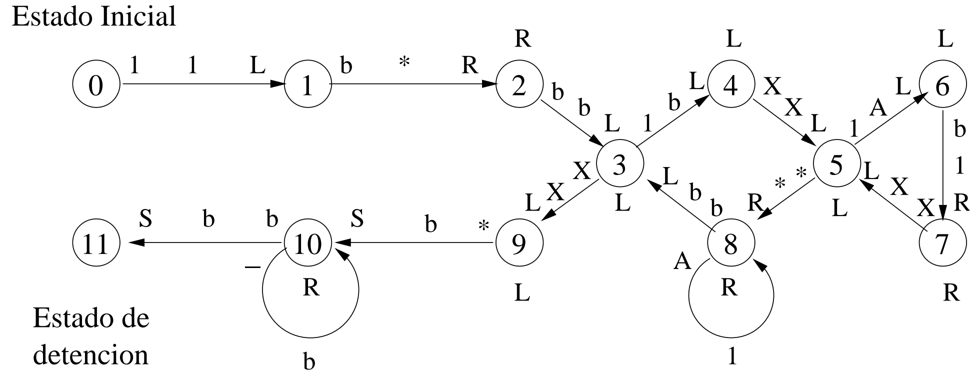


Figura 4.2: Multiplicando dos números unarios.

La primera acción de P, al ser confrontado con dos números unarios a multiplicar, es moverse una celda a la izquierda desde el 1 más a la izquierda y escribir un asterisco ahí, entrando al estado 2. El programa P permanece en el estado 2, contínuamente moviéndose a la derecha hasta encontrar la primera celda en blanco. En el estado 3, P cambia el 1 justo a este blanco (estado 3 $\rightarrow$ estado 4) y se mueve a la izquierda hasta encontrar un signo $\times$. Ahora, en el estado 5, entra en un ciclo de tres estados en el que cada 1 a la izquierda del número unario se cambia a un valor A, escribiéndose un 1 correspondiente en la primera celda disponible a la izquierda del asterisco. De esta forma, P escribe una caden de m unos. Cuando P finalmente encuentra el asterisco, la cadena se ha terminado y P pasa al estado 8, donde cambia todas las A otra vez a 1 y continúa moviéndose a la derecha hasta encontrar un blanco (estado 8 $\rightarrow$ estado 3). Aquí, el ciclo recomienza, el siguiente 1 en la cadena de n unos se borra, y P vuelve a entrar en el ciclo de tres estados a fin de escribir otra cadena de m unos adyacente a la cadena escrita anteriormente. De esta forma, n cadenas adyacentes de m unos se escriben, complentando la formación del producto $m \times n$ en forma unaria. En el paso final de este cómputo, P borra el asterisco (estado 9 $\rightarrow$ estado 10) y se mueve a la derecha, borrando todos los símbolos (lo que se denota con $-$) hasta encontrar el primer caracter blanco. Entonces se detiene.

Las máquinas de Turing fueron originalmente descritas por Alan Tu-

ring, uno de los fundadores de la computación moderna. Turing concibió su máquina a mediados de los 1930s en un esfuerzo por obtener en los términos más simples posibles lo que significa que un cómputo proceda en pasos discretos. Turing pensaba que su máquina era el modelo teórico concebible más poderoso, cuando observó que era equivalente a varios esquemas conceptuales de cómputo. Otro factor que le dió confianza en esta tesis fue la robustez de las máquinas de Turing. Sus alfabetos, conjuntos de estados y cintas pueden cambiarse radicalmente sin afectar el poder de la clase de máquinas que se obtenían.

Por ejemplo, el alfabeto de una máquina de Turing puede contener tan solo dos símbolos, y aún así, computar cualquier cosa que otros esquemas con alfabetos más grandes pueden computar. El número de estados puede reducirse también a sólo dos, sin alterar su potencia. Naturalmente, hay una relación entre el número de estados y el tamaño del alfabeto: dado un programa P arbitrario para una máquina de Turing empleando m estados y n símbolos, un programa basado en dos símbolos que imite a P debe, en general, tener muchos más que m estados. Similarmente, un programa con dos estados que imite a P debe tener muchos más que n símbolos.

Una de las características más robustas de las máquinas de Turing es la variación en el número y tipo de cintas que puede tener, lo que no afecta su poder de cómputo. Las máquinas de Turing pueden tener cualquier número de cintas, o pueden arreglárselas con tan solo una cinta semi-infinita. Esta no es más que una cinta infinita que ha sido cortada a la mitad, por así decirlo. Si se utiliza la mitad derecha, entonces no se le permite a la máquina de Turing moverse más allá de su celda más a la izquierda.

Un programa de una máquina de Turing multicinta no es ya una secuencia de quíntuplas, sino un conjunto de $(2+3n)$-tuplas, donde n es el número de cintas utilizadas. Las "intrucciones" de tal programa tienen la siguiente forma general:

$$q, s_1, s_2, \ldots, s_n, q', {s_1}', {s_2}', \ldots, {s_n}', d_1, d_2, \ldots, d_n$$

donde q es el estado actual, y $s_1, s_2, \ldots, s_n$ los símbolos que actualmente se leen (por medio de n cabezas lectoras/escritoras) de las n cintas. Bajo estas condiciones, esta $(2 + 3n)$-tuplas hace que la máquina entre en el estado q', escriba los símbolos ${s_1}', {s_2}', \ldots, {s_n}'$ en sus respectivas cintas y mueva las cabezas en las direcciones $d_1, d_2, \ldots, d_n$, donde cada d_i puede tomar los valores L, R y S, como se describe antes en la máquina de Turing de una sola cinta.

Aun cuando las máquinas multicinta no pueden computar aquéllo no computable por una máquina de una sola cinta, sin embargo, pueden computar de una manera mucho más eficiente. Por ejemplo, dada la tarea de formar el producto de dos números unarios, existe un programa para una máquina de tres cintas que lo realiza en aproximadamente $n \times m$ pasos. El programa que se presenta anteriormente requiere de $n^2 \times m^2$ pasos.

Finalmente, se demuestra que una máquina de Turing con una sola cinta puede hacer todo lo que una máquina con n cintas. Más precisamente, se muestra que dado el programa M para una máquina de Turing de n cintas, hay un programa P para una máquina de Turing con $(n-1)$ cintas que hace precisamente lo mismo. La construcción básica puede re-aplicarse para construir un equivalente con $(n-2)$ cintas, y continuar así hasta que finalmente resulta una máquina de Turing con una sola cinta y equivalente a M.

Por simplicidad, supóngase que M es una máquina de Turing con dos cintas. Denotando ambas cintas como $T1$ y $T2$, se selecciona una celda arbitraria en cada una de ellas como la celda 0, y el número de celdas a izquierda y derecha de estas celdas 0 se consideran numeradas con valores enteros positivos y negativos, respectivamente (figura 4.3).

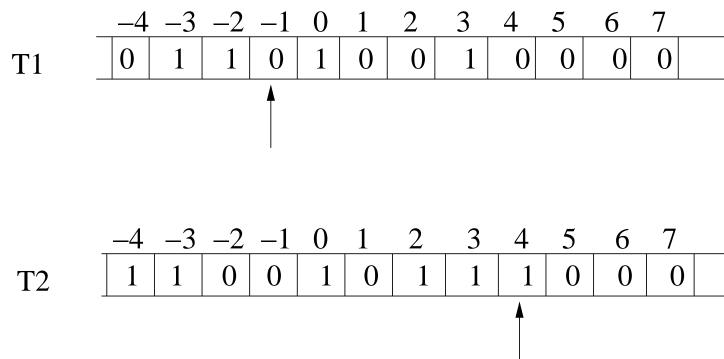


Figura 4.3: Las dos cintas de M.

En una nueva cinta, se numeran las celdas en pares $0,0$, $1,1$, etc. colocando el símbolo de la i-ésima celda de $T1$ en la primera celda del i-ésimo par, y colocando el símbolo de la i-ésima celda de $T2$ en la segunda celda del i-ésimo par (figura 4.4). Es conveniente suponer que el alfabeto en la

cinta de M consiste solo de número binarios 0 y 1. Las posiciones de las dos cabezas lectoras/escritoras se indican mediante los símbolos A y B. Estos símbolos sirven para indicar 0 y 1, respectivamente, es decir, cuál cabeza (1 ó 2) representan se mantiene por la máquina de Turing M' que se encuentra en consturcción.

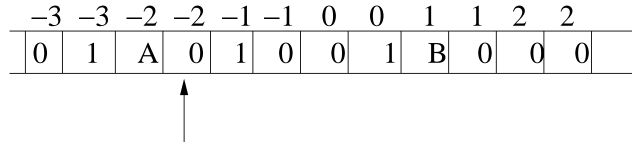


Figura 4.4: Juntando las dos cintas en una.

La idea general tras la construcción de M' es obtener una sola cabeza lectora/escritora que imite las cabezas de M mediante alternar entre los dos marcadores. Cada marcador, por lo tanto, se mueve dos celdas a la vez ya que dos celdas adyacentes en las dos cintas originales de M en M' se encuentran ahora separadas por dos celdas en M'. La nueva máquina de Turing M' se especifica en términos de un diagrama de transición de estados que tiene dos mitades simétricas. En una mitad (figura 4.5), el marcador de lectura y escritura para la cinta 1 se supone que se encuentra a la izquierda del marcador de lectura y escritura para la cinta 2. Si durante la simulación de M, los dos marcadores de M' se cruzan, una transición toma el control de la primera mitad a la segunda; aquí, el marcador de la cinta 1 se supone que se encuentra a la derecha del marcador de la cinta 2.

El primer paso en la operación de M' es decidir cuál óctupla de M debe "aplicarse" en seguida:

$$q, s_1, s_2, q', s_1{}', s_2{}', d_1, d_2$$

Dado que se supone que M se encuentra en estado q, se asigna el mismo estado a M', y se entra a una porción del diagrama de transición de estados que parece un árbol de decisión. La cabeza lectora/escritora de M' se encuentra leyendo actualmente el marcador izquierdo (figura 4.5).

En este diagrama y en los subsecuentes, se introducen algunas convenciones por simplicidad: si no aparece un símbolo a la mitad de una transición (flecha), meramente se re-escribe el símbolo al principio de la transición. Un estado etiquetado L ó R significa "movimiento a la izquierda" y "movimien-

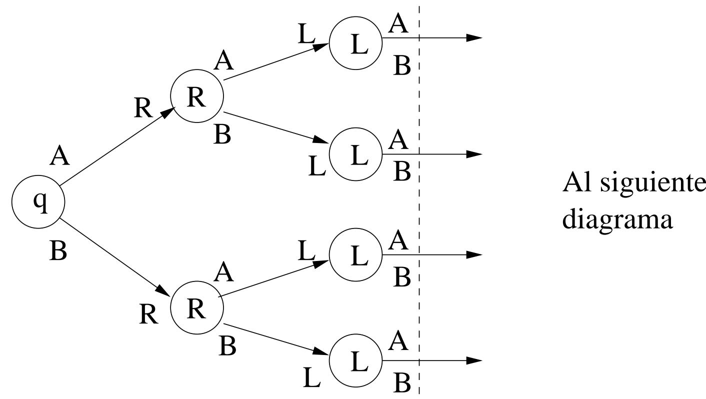


Figura 4.5: Leyendo los dos marcadores.

to a la derecha", respectivamente. Es decir, M' mantiene en movimiento su cabeza lectora/escritora en la dirección indicada, permaneciendo en ese estado hasta que se encuentra con un símbolo en una de sus transiciones actuales. Las transiciones con dos símbolos a su inicio en realidad representan dos transiciones, una por cada símbolo.

La máquina M' primero pasa a uno de sus dos estados de movimiento a la derecha dependiendo si el marcador de la derecha es A (0) ó B (1). Continúa moviéndose a la derecha hasta encontrar un marcador de la derecha. Aquí se hace una decisión similar, y la cabeza ahora se mueve de regreso al marcador izquierdo antes de una transición que sale del diagrama.

En este punto hay suficiente información para especificar qué transición se simula por M. Los símbolos s_1 y s_2 se conocen. Se representan por los valores (A ó B) de los dos marcadores. El diagrama que maneja la escritura de ${s_1}'$ y ${s_2}'$, así como los movimientos de los marcadores indicados por d_1 y d_2 tiene una estructura muy sencilla (figura 4.6).

En el marcador de la izquierda, M' reemplaza A ó B por una versión alfabética del símbolo ${s_1}'$, y entonces se mueve a la derecha hasta encontrar el marcador de la derecha. Este marcador se reemplaza por una versión alfabética de ${s_2}'$, y la cabeza de M' regresa al primer marcador, reemplazándolo por el correspondiente símbolo numérico. Entonces, M' se mueve un paso en la dirección d_1 y verifica si la celda actual contiene una A o una B. Si así es, los marcadores están a punto de cruzarse: una trasición apropiada

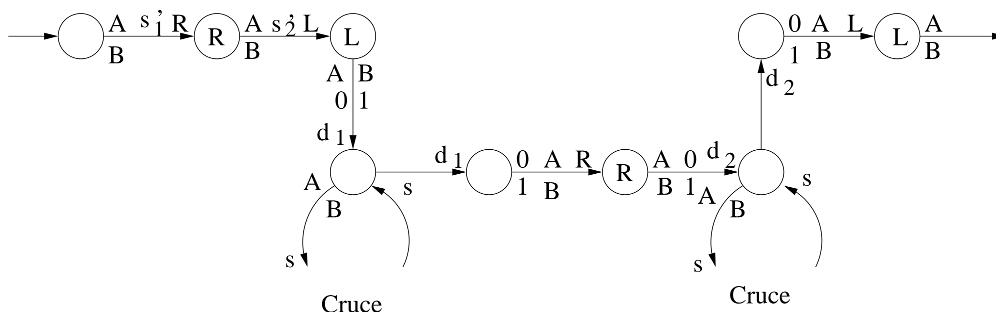


Figura 4.6: Escribiendo y moviendo los marcadores.

lleva entonces a M' a la otra mitad de su diagrama, aquél en el que el marcador 1 está a la derecha del marcador 2. Naturalmente, hay una transición correspondiente a este estado del estado correspondiente en la otra mitad del diagrama. En cualquier caso, M' hace otro movimiento en la dirección d_1, reemplazando el símbolo numérico en esta celda por su contraparte alfabética. Así, el marcador de la izquierda se ha movido dos celdas en la dirección d_1.

En seguida, M' realiza la operación correspondiente en el marcador de la derecha y finalmente regresa al primer marcador. Tan pronto como este marcador se encuentra, M' pasa por una transición que lo lleva fuera del diagrama anterior. ¿En qué estado entra M' enseguida? Entra a q' un estado "prestado" de M de la misma manera en que q está también prestado. Ahora M' está lista para pasar por otro árbol de decisión de estados como el primer diagrama.

Además de mostrar que las máquinas de Turing con n cintas no son más poderosas que las máquinas con una sola cinta, esta construcción resulta muy útil para la simplificación de mucho de la teoría de la computación mediante permitir confinar la discusión en cuanto a lo que las máquinas de Turing con una sola cinta pueden o no pueden hacer. Por ejemplo, se puede usar una máquina de tres cintas para simular un dispositivo que parece más poderoso. Al convertir esta máquina de tres cintas en su equivalente de una cinta por los métodos expuestos aquí, se puede obtener una máquina de Turing con una sola cinta que realiza la simulación del dispositivo.

Capítulo 5

Máquinas Universales de Turing

Computadoras como Programas

Las máquinas de Turing se describen como un tipo de computadora abstracta. Sin embargo, tal terminología puede ser en ocasiones equivocada, ya que las máquinas de Turing normalmente se definen para resolver problemas específicos, y no son programables en el sentido común de la palabra. Es cierto que, si se piensa en una máquina de Turing sólo en términos de su *hardware*, es decir, en la cinta, la cabeza lectora/escritora, la unidad de control, y demás partes, entonces es razonable distinguir entre una máquina de Turing y los programas que se ejecutan en ella. Tales programas pueden darse como entrada a la máquina de Turing en forma de quíntuplas.

En un sentido, la descripción abstracta de una máquina de Turing programable ya se ha realizado. Se le conoce como "Máquina Universal de Turing" (*Universal Turing Machine*), y fue inicialmente definida por el propio Alan Turing en los años 1930 como un modelo abstracto de un dispositivo de cómputo del tipo más general que pudo imaginar.

Una máquina universal de Turing (figura 5.1) tiene un programa fijo permanentemente grabado en su unidad de control. Este programa simula la acción de una máquina de Turing arbitaria (o un programa) mediante leer el programa en una cinta y simular su comportamiento en otra.

La máquina universal de Turing U que se describe aquí normalmente cuenta con dos cintas. La cinta 1 contiene una lista de quíntuplas que definen

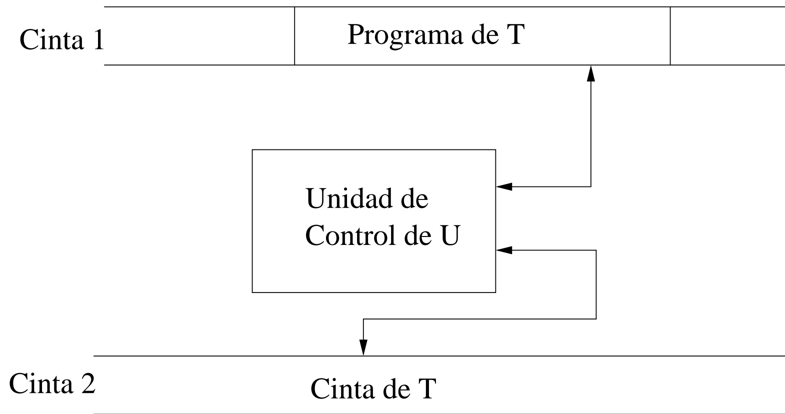


Figura 5.1: Una máquina universal de Turing.

a una máquina de Turing T, y la cinta 2 se encuentra en blanco. En esta última cinta, U simula la acción de T: las apariencias inicial y final de la cinta 2 son idénticas a las que resultarían si T hubiera operado directamente en ella.

Se sabe que una máquina de Turing con dos cintas puede simularse mediante una máquina de Turing de una sola cinta, y esta observación se aplica con la misma intensidad a las máquinas universales de Turing. En otras palabras, dada la contrucción de U, sería una cuestión sencilla la construcción de una máquina universal de Turing de una sola cinta U^1 que la simulara.

El programa fijo que habita en la sección de control de U es realmente análogo a un intérprete de software en una computadora digital. Se divide en dos secciones:

1. Dados el estado actual de T y un símbolo de entrada, encuentra la quíntupla (q, s, q', s', s) en la descripción de T que aplique.

2. Registra el nuevo estado q', se escribe el nuevo símbolo s' en la cinta 2, se mueve la cabeza 2 en la dirección d, se lee el nuevo símbolo en la cinta 2, y se registra junto con q'.

La máquina universal U espera un formato particular para la descripción del programa de T. Las quíntuplas de T y todos los símbolos de la cinta de T usan el alfabeto binario. Así, si T tiene n estados, entonces números binarios de k bits se utilizan para indexar los estados de T, donde $k = \lceil logn \rceil$. Dados

dos movimientos básicos de las cintas, izquierdo (igual a 0) y derecho (igual a 1), cada quíntupla puede listarse como un número binario de $(2k + 3)$ bits. Las quíntuplas se separan por los caracteres X y Y como marcadores de límites a ambos lados de las quíntuplas de T. En la quíntupla que se muestra en la figura 5.2, los bits señalados con q y s representan el estado actual y símbolo de entrada de T, y q' y s' representan el nuevo estado y en nuevo símbolo de T, mientras que d representa la dirección en la que la cabeza de T se debe mover.

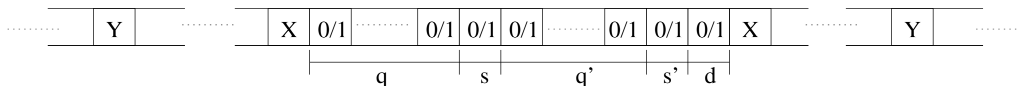


Figura 5.2: Un programa a ejecutarse.

Ls figura 5.3 muestra una porción del programa intérprete de U, en forma de un diagrama de transición de estados. Se ha compactado el diagrama mediante designar cada estado para moverse a la derecha (R) o a la izquierda (L); toda transición de salida de esos estados (y posiblemente, retornando a ellos) involucra el movimiento de la cabeza de U en la dirección correspondiente. Muchas transiciones por las que pasa U en este y subsecuentes diagramas no se muestran. Una transición "perdida" simplemente significa que U permanece en el mismo estado, moviéndose de acuerdo con la dirección del estado y escribiendo cualquier cosa que lea. El propósito de esta porción que se muestra del diagrama de U es localizar la siguiente quíntupla a ejecutarse mediante encontrar sus primeros dos elementos, el estado q y el símbolo s.

La máquina U utiliza $(k - 1)$ segmentos de bit de la cinta 1 inmediatamente a la izquierda del marcador Y izquierdo como espacio de trabajo para registrar la cinta de T y el símbolo de entrada. Dado que una etiqueta de estado q y un símbolo s ocupan este espacio, el programa para localizar q/s es fácil de describir. Con la cabeza lectora/escritora 1 en el marcador Y de la izquierda, U inicia el estado con la transición de "Inicio". Revisa hacia la izquierda, cambiando cada 0 por A y cada 1 por B hasta encontrar un caracter blanco b. Entonces se mueve a la derecha, cambiando el primer símbolo no-blanco que encuentre a 0 ó 1 dependiendo de si el símbolo era una A o una B. Entonces, se mueve a la derecha, buscando una coincidencia con ese símbolo. El primer bit 0 ó 1 encontrado dispara ya sea un retorno al mismo ciclo de coincidencia o regresa al estado de inicio, dependiendo

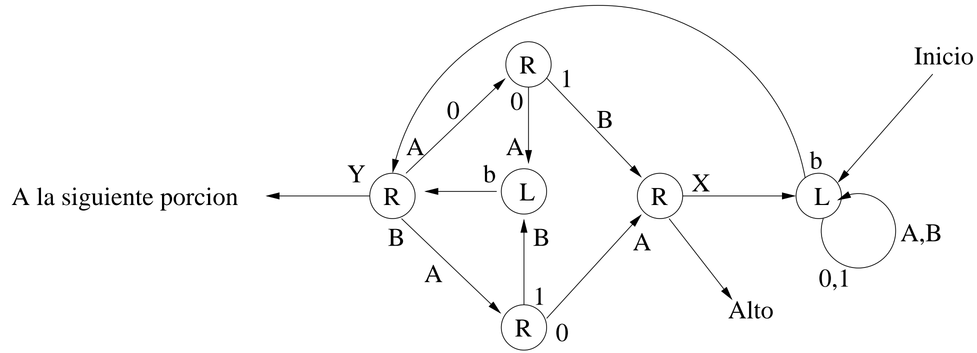


Figura 5.3: Búsqueda de la quíntupla actual.

si se ha encontrado una coincidencia. En cualquier caso, la porción q/s del primer símbolo no coinciden con la q y la s almacenados en el espacio de trabajo. En tal caso, U se mueve a la derecha a la primera X y entonces re-entra al estado inicial, donde cambia todos los ceros y unos entre esa X y el extremo final izquierdo del espacio de trabajo, a A y B respectivamente. Entonces U intenta hacer coincidir la q y la s en su espacio de trabajo con la siguiente quíntupla. Finalmente, U tiene éxito, pasando mediante una transición al siguiente diagrama, o deteniéndose por no encontrar ninguna coincidencia. En tal caso, T también se detendría bajo la convención usual para las máquinas de Turing. La figura 5.4 muestra la cinta 1 después de que U ha encontrado una coincidencia para q y s en la segunda quíntupla.

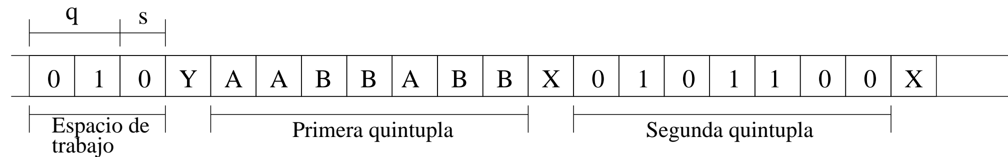


Figura 5.4: U localiza la quíntupla.

La segunda porción del programa de U registra el nuevo estado q' en el espacio de trabajo, y mueve la cabeza lectora/escritora 2 de acuerdo con la quíntupla actual de T. Entonces, registra el nuevo símbolo s' en el espacio de trabajo junto a q' en la cinta 1. En la figura 5.5, los estados representados en forma cuadrada indica movimientos de la cabeza sobre la cinta 2.

Correspondientemente, los ceros y unos subrayados indican símbolos leidos o escritos en la cinta 2.

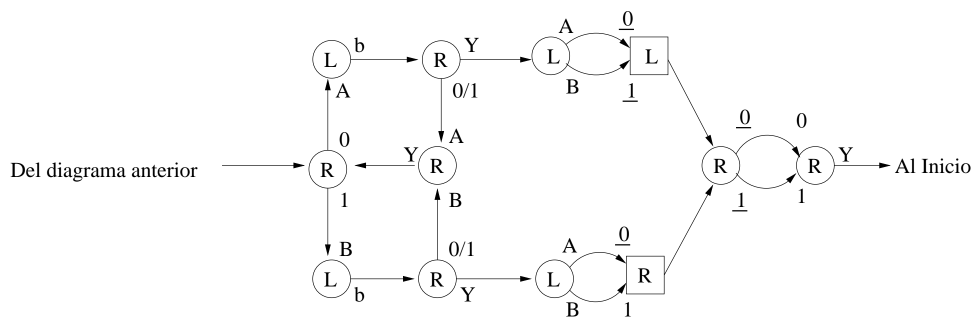


Figura 5.5: Registrando el nuevo estado y moviendo la cabeza de T.

Esta parte del programa de U se invoca cuando la primera porción ha localizado la quíntupla cuyo componente q/s coincide con el contenido del espacio de trabajo. Tal componente ha sido re-escrito en términos de A y B, pero el componente q'/s' aún existe como una cadena de ceros y unos. Esta porción del programa de U comienza con la cabeza 2 justo a la derecha del marcador izquierdo Y.

Revisando hacia la derecha, los primeros símbolos binarios que U encuentra pertenecen a el siguiente estado q'. Estos símbolos se copian uno a la vez, en términos de A y B, en el espacio de trabajo. Cuando esta sección del programa de U ha terminado de copiar q', enseguida encuentra el único símbolo binario de s' en la quíntupla de T. La máquina U copia s' como una A o una B en la última celda, la que se encuentra más a la derecha del espacio de trabajo, revisando hacia la derecha de la quíntupla para recoger el último símbolo, que corresponde a d.

En la figura 5.6 se muestra la cinta 1 en este momento de la operación de U. Cuando U retorna al espacio de trabajo, sin embargo, encuentra que se ha terminado el espacio: encuentra el marcador Y. En este punto, U "recuerda" el valor de d mediante encontrarse en la parte superior o inferior de su diagrama de estados. En cualquier caso, U revisa hacia la izquierda a fin de recoger el símbolo s' como una A o una B, convierte éste en el correspondiente 0 ó 1, y lo escribe en la cinta 2 tal y como T lo haría. Finalmente, U mueve la cabeza 2 a la izquierda (en la parte superior del diagrama) o a la derecha (en la parte inferior del diargrama), como lo haría

T, y entonces lee el siguiente símbolo de la cinta 2. Reemplaza s' en el espacio de trabajo, U se recorre hasta el marcador Y y regresa a la porción del programa de la figura 5.3.

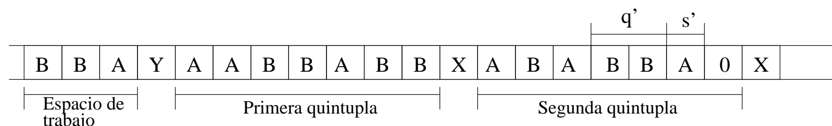


Figura 5.6: ¿Para qué lado se mueve la cabeza de T?.

Para iniciar a U al principio de la ejecución del programa de T, es necesario colocar la cabeza 1 sobre el marcador Y izquierdo. La cabeza 2 se coloca sobre la celda inicial de la cinta de T. El espacio de trabajo debe inicializarse también mediante colocar una cadena de k bits para el estado q_0 (el estado inicial de T) junto con el símbolo que se lee inicialmente en la cinta 2. Entonces, el programa de U hace el resto.

La tesis de Turing es un paralelo muy cercano a la tesis de Church, en cuanto a declarar que cualquier cosa que razonablemente podría significar un "procedimiento efectivo" se captura por un esquema computacional específico, en este caso, las máquinas de Turing. Una máquina universal de Turing incorpora la tesis de Turing de una sola vez, por decirlo de algún modo, mediante representar simultáneamente todas las máquinas de Turing posibles y (por la tesis de Turing) todos los procedimientos efectivos. Lo hace en un nivel abstracto, de una manera muy parecida a cómo una computadora digital de propósito general representa todos los posibles programas para ella misma. Su propia existencia es un reto para explorar el rango de programas posibles; qué puede hacer y qué no puede hacer.

En la teoría de la computación, la noción de las máquinas universales de Turing sirve como una aproximación para responder algunas preguntas concernientes a la existencia de procedimientos efectivos. Por ejemplo, el problema de la detención (*halting problem*) pregunta si hay un procedimiento efectivo que decida, para cada par posible (T, t), si T finalmente se detiene en t. En lugar de construir tal procedimiento para todos los pares (T, t), es tan solo necesario construir uno para (U, t), ya que t puede inicialmente contener T, y por lo tanto, U se detiene si y solo si T se detiene.

Capítulo 6

Cálculo de Predicados
El Método Resolución

El cálculo de predicados es uno de los lenguajes más poderosos conocidos para la expresión de ideas y pensamientos matemáticos. Ha influenciado a la Computación directamente a través de la invención de los lenguajes de programación como LISP, e indirectamente a través de las teorías de la computación que dependen de él.

Como un ejemplo simple del poder expresivo del cálculo de predicados, considérese el conocido problema del lobo, la cabra y el heno: un hombre desea llevar a un lobo, una cabra y una paca de heno al otro lado de un río. Para ello, cuenta con un bote de remos, pero éste es demasiado pequeño, tanto que sólo puede transportar a alguno de los tres seres además de sí mismo. El problema es que no puede dejar al lobo y a la cabra en una sola ribera mientras cruza el río, ya que el lobo se comería a la cabra. De manera similar, no puede dejar a la cabra con el heno, ya que la cabra se comería el heno.

Muchas configuraciones posibles pueden utilizarse que combinan al hombre, el lobo, la cabra y el heno en su travesía por el río. Todas ellas pueden representarse mediante un solo predicado P. Es en realidad una función de verdad, que cuenta con cuatro argumentos: m, w, g y c, los cuales toman valores binarios. Por ejemplo, $m = 1$ significa que el hombre está en la ribera inicial, mientras que $m = 0$ significa que está el otro lado. Una convención similar puede considerarse para w (el lobo), g (la cabra) y c (el heno). El predicado se escribe funcionalmente como:

$$P(m, w, g, c)$$

y tiene una intepretación muy específica: para cada posible combinación de valores de las cuatro variables lógicas, P es verdadero para estas variables si y solo si la configuración correspondiente puede realizarse sin que el lobo se coma a la cabra ni que la cabra se coma el heno. Este ejemplo se vuelve a toma más adelante, para plantear su solución.

El interés de la Computación en el cálculo de predicados ha sido explotar su poder de expresividad para comprobar teoremas. En el campo de la Inteligencia Artificial, se han hecho varios intentos para construir programas que puedan comprobar teoremas automáticamente. Dado un conjunto de axiomas y una técnica para derivar nuevos teoremas a partir de teoremas comprobados, ¿podría uno de estos programas comprobar un teorema particular que se le provea? Algunos intentos iniciales fallaron porque no había una técnica lo suficientemente eficiente para derivar nuevos teoremas. Hasta que en 1965, J.A. Robinson, de la Universidad de Syracuse, descubrió una técnica llamada "resolución" (*resolution*). No sólamente resolución permite frecuentemente la derivación de teoremas, sino también fue la base de la "quinta generación" de computadoras, las cuales se esperaba dedicaran su tiempo a comprobar teoremas. Estos teoremas se consideran como problemas de recuperación y deducción de hechos sobre bases de datos. De hecho, el problema de que el hombre exitosamente logre cruzar con el lobo, la cabra y el heno puede expresarse como un teorema.

Sintácticamente, el cálculo de predicados puede definirse recursivamente como cadenas de símbolos que siguen una manera particular de construcción: tal y como se define el cálculo proposicional.

Los bloques básicos de construcción de fórmulas en el cálculo de predicados son los símbolos individuales. En la aproximación algo informal que se presenta aquí, las letras mayúsculas denotan predicados mientras que las minúsculas indican funciones y variables. Se entiende que si se requieren más predicados, funciones o variables, es posible utilizar la subscripción de las letras. Otra convención utilizada es escribir los números en su forma común, en lugar de escribirlos en notación unaria.

Además de los símbolos alfabéticos, se requieren paréntesis, y una colección de operadores lógicos estándar como $\vee$, $\wedge$, $\rightarrow$ y $\sim$, así como otros dos símbolos muy peculiares dentro del cálculos de predicados: los cuantificadores existencial ($\exists$) y universal ($\forall$).

Un "término" se define recursivamente como sigue:

1. Un término es una variable.
2. Si f es una función de n argumentos, y $x_1, x_2, \ldots x_n$ son términos, entonces $f(x_1, x_2, \ldots x_n)$ es un término.

Una "fórmula atómica" se define como cualquier predicado cuyos argumentos son términos. Finalmente, el objetivo de estas definiciones, una "fórmula", se define recursivamente como:

1. Una fórmula atómica es una fórmula.
2. Si F y G son fórmulas, también los son $(F \vee G)$, $(F \wedge G)$ y $(\sim F)$.
3. Si F es una fórmula y v es una variable, entonces $\forall v(F)$ y $\exists v(F)$ son fórmulas.

Nótese en estas definiciones el uso de símbolos como x_1 para términos y F para fórmulas; éstos no son elementos del cálculo de predicados, sino en realidad parte de un metalenguaje usado para definirlo. Un ejemplo de una fórmula en el cálculo de predicados (que a partir de aquí se le llama simplemente "fórmula") es:

$$\forall e(P(e) \rightarrow (\exists d(P(d) \wedge (L(a(x,y),d) \rightarrow L(a(f(x),f(y)),e)))))$$

Es tal vez un poco injusto esperar que un estudiante de teoría de funciones reconozca en esta fórmula la definición de continuidad en una función real. En el estilo informal de los textos de matemáticas, esta definición se escribe normalmente como sigue:

$$\forall \epsilon > 0 \, \exists \delta > 0 \text{ tal que } \mid x - y \mid < \delta \rightarrow \mid f(x) - f(y) \mid < \epsilon$$

En el contexto más restrictivo del cálculo de predicados, esta definición debe re-formularse en términos de variables, funciones, predicados, y sus combinaciones legales en fórmulas. Por ejemplo, $\forall \epsilon > 0$ se escribe como $\forall e(P(e) \rightarrow \ldots)$. Aquí, $P(e)$ es un predicado cuyo único argumento es una variable real. Se interpreta para significar que e es un número positivo. Si e es positivo, la fórmula continúa para decir que hay un d tal que d es positivo y que la fórmula

$$L(a(x,y),d) \rightarrow L(a(f(x),f(y)),e)$$

es verdadera. Aquí, L es un predicado que es verdadero cuando su primer argumento es menor que el segundo. La función a define el valor absoluto de la diferencia de sus argumentos. Bajo la interpretación dada a la fórmula como un todo, obviamente expresa la continuidad de una función real.

Por supuesto, tal fórmula no es en realidad equivalente a la definición de continuidad hasta que otras fórmulas adicionales que definen la noción de número, inequidad, etc. son dadas. Ciertamente, los libros más rigurosos sobre funciones reales hacen algo cercano a esto. En cualquier caso, no sólo puede la teoría de funciones proponerse en el cálculo de predicados, sino virtualmente todas las matemáticas. Este es el objetivo del *Principia Mathematica*, la ambiciosa codificación lógica de las matemáticas realizado en 1921 por los matemáticos británicos Bertrand Russell y Alfred North Whitehead.

El cálculo de predicados hereda muchas de las reglas de manipulación del cálculo proposicional. Por ejemplo, las leyes de De Morgan establecen que:

$$\sim (A \vee B)$$

es lógicamente equivalente a:

$$(\sim A) \wedge (\sim B)$$

donde A y B son proposiciones. Si se reemplazan A y B por predicados arbitrarios, la regla aún se mantiene. Esto también sucede con reglas adicionales de manipulación relativas a los cuantificadores universal y existencial. Por ejemplo:

$$\sim (\forall x(P(x)))$$

es equivalente a:

$$\exists x(\sim (P(x)))$$

La regla también funciona a la inversa, con:

$$\sim (\exists x(P(x)))$$

equivalente a:

$$\forall x(\sim (P(x)))$$

Es necesario distinguir en las fórmulas entre variables "libres" y variables "dependientes". La dependencia en este caso se refiere a estar bajo un cuantificador particular. Más precisamente, si F es una fórmula y x es una variable en F, entonces para ambos $\forall x(F)$ y $\exists x(F)$ la variable es dependiente. Cualquier variable en F que no se cuantifique es llamada libre.

Como en el cálculo proposicional, algunas fórmulas predicadas son verdaderas, y otras falsas. Su veracidad depende, sin embargo, en cómo sus predicados y funciones se interpretan. Una interpretación de una fórmula involucra un universo U y una función de interpretación I que mapea cada predicado n-localizado de la fórmula en una relación n-aria en U. También mapea cada símbolo de la función n-localizada de la fórmula en una función n-localizada de U en sí mismo.

Si la fórmula contiene una variable libre, en general es imposible bajo cualquier interpretación de F determinar su veracidad. Por ejemplo, bajo la usual interpretación del predicado menor-que L, no se puede decir si la fórmula

$$\forall x L(x, y)$$

es verdadera o no. Obviamente, lo sería si se inserta $\exists y$ inmediatamente después de $\forall x$. En cualquier caso, una fórmula que no contiene variables libres es "satisfactible" (*satisfiable*) si tiene al menos una interpretación en la cual es verdadera. Se le llama "válida" si es verdadera bajo todas las posibles interpretaciones.

Determinar la validez de una fórmula no es una cosa sencilla. Aun decidir si es satisfactible resulta irresolvible: no hay un procedimiento efectivo que decida la satisfactibilidad de fórmulas arbitrarias.

Hay procedimientos, sin embargo, para derivar fórmulas de otras fórmulas. Las "otras" fórmulas pueden ser llamadas "axiomas", y la fórmula derivada puede llamarse un "teorema". Derivar un teorema de un conjunto de axiomas ha sido la labor de los matemáticos desde el tiempo de Euclides, y mucho antes.

Para la comprobación automática de teoremas en computación, el procedimiento seleccionado es el método resolución. En él, los axiomas con los que se inicia tienen una cierta forma, llamada "clausal". Una "cláusula" es simplemente una disyunción de predicados o sus negaciones. Cualquier fórmula en el cálculo de predicados puede escribirse en esta forma.

Considérense dos cláusulas $(\sim P(x) \vee Q(x,y) \vee R(y))$ y $(P(x) \vee \sim S(x))$. En una de estas cláusulas el predicado $P(x)$ aparece, y en la otra se ve su negación. Dado que no hay otro predicado que comparta esta propiedad respecto a las dos cláusulas, se les puede reemplazar por la disyunción de todas las literales tomadas en conjunto, excluyendo $P(x)$ y $\sim P(x)$:

$$(Q(x,y) \vee R(y) \vee \sim S(x))$$

Desafortunadamente, resolución no es tan directo en muchos otros casos. Por ejemplo, el predicado que se elimina puede tener una estructura más complicada. Supóngase que $P(a, f(x))$ aparece en una cláusula y que $P(x,y)$ aparece en la otra. "Unificar" tales predicados significa encontrar una substitución para las variables que ocurren en ellas, que las exprese precisamente en la misma forma. Esta substitucion debe también ser la más general posible al efectuar este propósito, en lugar de encontrar el mínimo denominador común. La substitución para el ejemplo dado anteriormente sería $x = a$ y $y = f(a)$. Realizando la substitución, resulta en la aparición de:

$$\sim P(a, f(a)) \;\text{ y }\; P(a, f(a))$$

En este punto, resolución puede llevarse a cabo como antes.

Supóngase ahora que F es un teorema a ser comprobado y que $A_1, \ldots, A_n$ son todos axiomas. Formalmente hablando, F y $A_1, \ldots, A_n$ son todos fórmulas clausales en el cálculo de predicados. Los cuantificadores existenciales se han removido por un proceso de "skolemización" (*skolemization*): si $\exists x$ aparece en una fórmula, se reemplazan las ocurrencias de x limitadas por su cuantificador por una instancia particular $x = a$ que hace verdadera a la expresión bajo $\exists$. Cada variable de cada cláusula se entiende como universalmente cuantificable, de modo que los signos $\forall$ se omiten. El método resolución procede mediante primero negar el teorema F y entonces adjuntarlo con los axiomas. El sistema resultante puede escribirse como:

$$\sim F, A_1, A_2, \ldots, A_n$$

A grandes rasgos, el método consiste en resolver pares de cláusulas dentro de este sistema hasta que se llegue a una contradicción. Ambos, un predicado y su negación, son resultado de resolución. Si esto sucede, F ha sido comprobado. Si esto no puede hacerse, F no puede comprobarse a partir de los axiomas.

Recordando el predicado $P(m, w, g, c)$, el universo en el que se interpreta éste consiste de un río, un hombre, un lobo, una cabra y una paca de heno. Las variables m, w, g y c son valuadas binariamente, y se refiere a la situación de un lado u otro del río: si $m = 0$, el hombre está en el margen inicial del río. Si $m = 1$ el hombre está del otro lado. De tal modo, considerando el lobo, la cabra y el heno, el predicado

$$P(0, 1, 1, 0)$$

es verdadero si es posible para el hombre y el heno estar en el lado inicial del río y el lobo y la cabra del otro. Sería desafortunado para la cabra si esto fuera verdadero.

Además, resulta útil contar con un predicado de equidad $E(x, y)$ que es verdadero si x y y tienen el mismo valor. Una función f hace posible distinguir uno de los lados del río del otro: $f(0) = 1$ y $f(1) = 0$. En general, se usa f para definir los primeros axiomas, que regulan la equidad:

1. $(E(x, x))$
2. $(\sim E(x, f(x)))$

Las operaciones de cruce permisibles arrojan otros cuatro axiomas. Sólo se explica aquí la derivación del primero.

Si el hombre y el lobo estuvieran del mismo lado del río, entonces es posible que el hombre lleve al lobo al otro lado, resultando que la cabra y el heno se quedan del otro lado. Esto puede escribirse en términos de la función y los predicados como sigue:

$$P(x, x, y, z) \wedge \sim E(y, z) \rightarrow P(f(x), f(x), y, z)$$

Reescribiendo en forma clausal, esto se convierte en el siguiente axioma:

3. $(\sim P(x, x, y, z) \vee E(y, z) \vee P(f(x), f(x), y, z))$

Las operaciones de cruce restantes son:

4. $(\sim P(x, y, x, z) \vee P(f(x), y, f(x), z))$
5. $(\sim P(x, y, z, x) \vee E(y, z) \vee P(f(x), y, z, f(x)))$
6. $(\sim P(x, y, x, y) \vee E(x, y) \vee P(f(x), y, x, y))$

Otros dos axiomas se requieren para completar de la solución. El primero representa la condición inicial del hombre, el lobo, la cabra y el heno. El segundo respresenta la condición final. Esta cláusula está negada:

7. $(P(0,0,0,0))$
8. $(\sim P(1,1,1,1))$

En seguida se muestra un conjunto de unificaciones y resoluciones como sigue:

1. Comenzando con el axioma **4.** cuando $x = y = z = 0$ resulta en $(\sim P(0,0,0,0) \vee P(1,0,1,0))$. Considerando la condición inicial **7.**, la expresión anterior puede reducirse a:

$$(P(1,0,1,0)) \; *$$

2. Por otro lado, se tiene el axioma **6.** cuando $y = f(x)$, resultando en $(\sim P(x, f(x), x, f(x)) \vee E(x, f(x)) \vee P(f(x), f(x), x, f(x)))$. Este resultado se reduce junto con el axioma **2.**, dando:

$$(\sim P(x, f(x), x, f(x)) \vee P(f(x), f(x), x, f(x)))$$

Evaluando esto último en $x = 1$, resulta:

$$(\sim P(1,0,1,0) \vee P(0,0,1,0))$$

3. Finalmente, reduciendo $(P(1,0,1,0)$ y $(\sim P(1,0,1,0) \vee P(0,0,1,0))$ se obtiene:

$$(P(0,0,1,0)) \; *$$

Las dos cláusulas marcadas con asteriscos de los pasos 1 y 3 representan dos partes de la solución del problema del cruce del río. En la primera, el hombre ha tomado a la cabra al otro lado del río. En la segunda ha regresado al lado original.

Los pasos de resolución se han seleccionado teniendo la solución en mente. Un sistema real de comprobación de teoremas no tendría tal comportamiento. Procedería mas bien a ciegas, buscando por resoluciones y unificaciones, construyendo una lista de cláusulas intermedias hasta finalmente derivar $(P(1,1,1,1))$. Comparando esto con el axioma **8.**, obtendría una contradicción. Este hecho se garantiza por un teorema de Robinson, que establece que resolución llega a una contradicción si y solo si la fórmula originalmente negada es lógicamente derivada de todos los axiomas.

Hay numerosas estrategias que apoyan al método resolución para obtener una contradicción tan rápido como sea posible. Una se conoce como "estrategia de preferencia de unidad" (*unit preference strategy*): seleccione aquellas cláusulas (para unificar y resolver) que sean tan cortas como sea posible. Otra estrategia, llamada "conjunto de apoyo" (*set-of-support*) mantiene una distinción entre axiomas primarios y de apoyo: ningún par de axiomas primarios se resuelven en contra entre sí.

El método resolución se vuelve muy poco manejable, desde el punto de vista computacional, para problemas más complicados que el problema del cruce del río. Sin embargo, deducciones así de complicadas no aparecen comúnmente en el contexto de muchas aplicaciones de programación lógica.

Capítulo 7

El Problema de la Detención *Lo No-Computable*

La máquina de Turing es una de varias formulaciones equivalentes de lo que significa un "procedimiento efectivo" o "cómputo". Nada más poderoso que una máquina de Turing ha sido descubierto que capture mejor el significado de tales términos. Y aún así, hay límites para el poder de las máquinas de Turing, problemas que estas máquinas no pueden resolver. Tales problemas son semejantes a aquéllos en los que no existe un procedimiento efectivo o cómputo recursivo que los resuelva. De hecho, se puede reformular un problema no-resoluble por una máquina de Turing como un problema no-resoluble en cualquier sistema formal equivalente.

El problema más conocido no-resoluble por máquinas de Turing es el "problema de la detención" (*halting problem*). En él, se requiere una máquina de Turing T_H (figura 7.1) capaz de realizar la siguiente tarea para cualquier par (T, t) como entrada,

> Dada una máquina de Turing arbitraria T como entrada y una cinta igualmente arbitraria t, decidir si T se detiene con t.

Naturalemente, T debe ser del tipo de máquinas de Turing que ejecuta sobre una cinta semi-infinita, pues al dar como entrada el par (T, t) a T_H, una mitad de la cinta de T_H debe contener la descripción de T, llamada dT, y la otra mitad es un duplicado de la cinta semi-infinita t. Un esquema similar se usa para implementar una máquina universal de Turing.

La pregunta es: ¿existe tal máquina T_H?

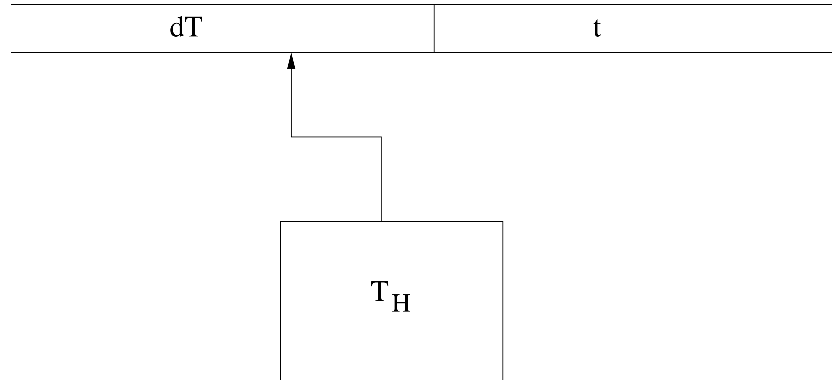


Figura 7.1: Una máquina de Turing que resuelve el problema de la detención.

Supóngase por un momento que sí. Si T se detiene con t, entonces tarde o temprano T_H señalará el equivalente a un "sí", y al hacerlo, completa una transición de algún estado q_i a un estado de detención q_h (figura 7.2). Si T no se detiene con t, sin embargo, entonces T_H tarde o temprano dirá "no", haciendo otra transición de un estado q_j a un estado de detención q_k (figura 7.3).

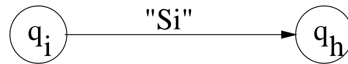


Figura 7.2: Transición para un "sí".

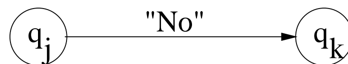


Figura 7.3: Transición para un "no".

Ahora bien, mediante realizar algunas alteraciones simples a T_H, se pueden complicar las cosas seriamente. La primera alteración resulta de preguntar si T_H puede decidir si T se detiene con dT en lugar de con t (figura 7.4).

Si se le solicita a T_H realizar esta tarea especializada (o más bien extraña), se le podría proveer a T_H con una cinta más compacta que contenga

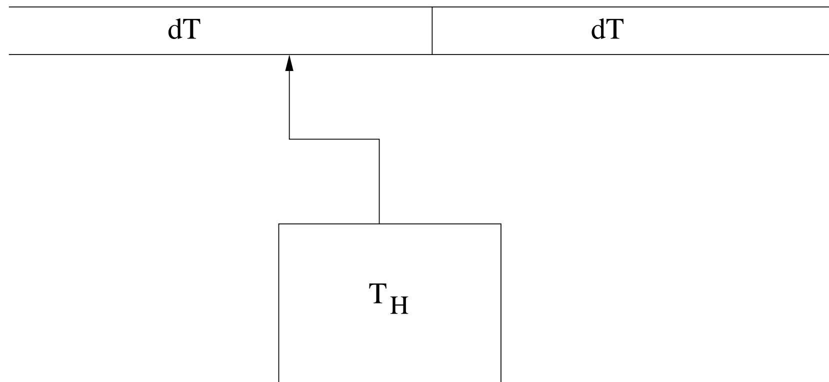


Figura 7.4: ¿Se detiene T con su propia descripción?

sólo una copia de dT, mediante implantar una máquina de Turing especial T_C dentro de T_H. El objetivo de T_C es hacer una copia de dT y, cuando haya terminado, entregar los datos a T_H mediante una transición del estado final de T_C al estado inicial de T_H (figura 7.5).

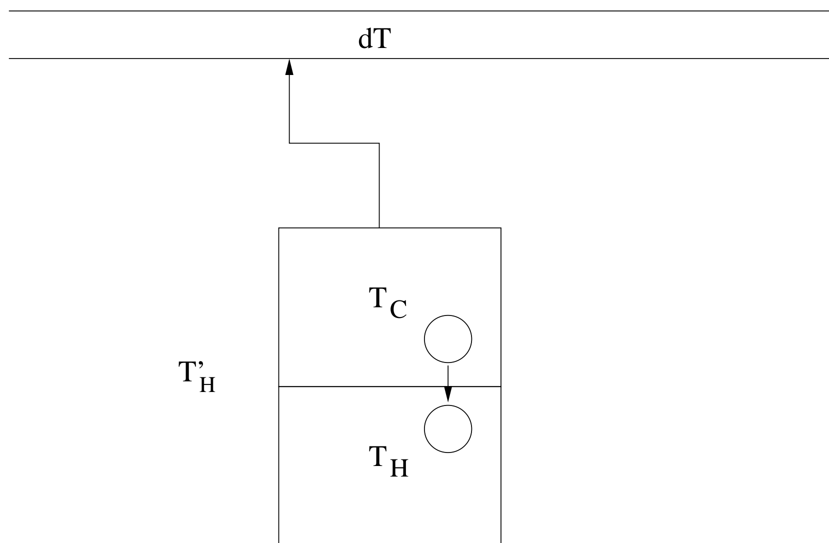


Figura 7.5: Una máquina de copia se incluye en T_H.

Denotando la máquina resultante como T'_H, se realizan algunos cambios

leves sobre sus dos transiciones de detención. La transición para "sí" desde q_i y la transición para el "no" desde q_j se desvían a dos nuevos estados, q_n y q_y, respectivamente (figura 7.6). Una vez que se llegue al estado q_n, hay una transición hacia q_n para todo posible combinación de estado/entrada en la cual T'_H pueda encontrarse. Así, una vez en el estado q_n, la máquina de Turing resultante T''_H nunca se detiene. Sin embargo, una vez en el estado q_y, T''_H se detiene por definición: q_y es un estado de detención (figura 7.7).

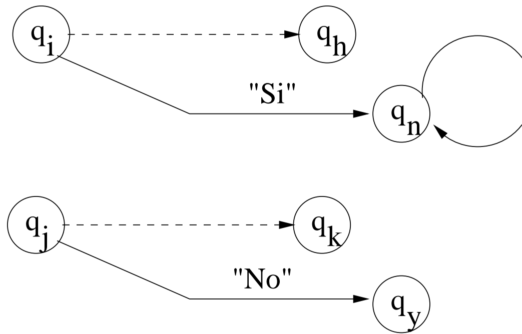


Figura 7.6: Recolocando los estados de detención de T_H.

Ahora bien, se puede hacer una prueba muy interesante si se alimenta la cinta dT''_H a T''_H para procesarla. Si T''_H se detiene con dT''_H, entonces debe tomar la misma transición para el "sí" que T_H debe tomar. Pero al hacerlo, entra en un estado en el que debe estar en ciclos infinitos, y nunca se detiene: si T''_H se detiene con dT''_H, entonces T''_H *no* se detiene con dT''_H. La situación no mejora si se supone que T''_H no se detiene con dT''_H. Para tal caso, dT''_H debe tomar la transición "no", terminando en el estado q_y, un estado de detención: si T''_H no se detiene con dT''_H, entonces T''_H se detiene con dT''_H.

Estas contradicciones aseguran que la máquina T_H no puede existir desde un principio. Así, el problema de la detención no puede resolverse por ninguna máquina de Turing.

Un problema similar que no puede resolverse es el siguiente: ¿hay una máquina de Turing, la cual, para cualquier par dado (T, t) como entrada, puede decidir si T imprime el símbolo x cuando procesa la cinta t? Ciertas alteraciones en la máquina T_p que se supone puede resolver el "problema de la impresión" (*printing problem*) resultan en el mismo tipo de contradicciones.

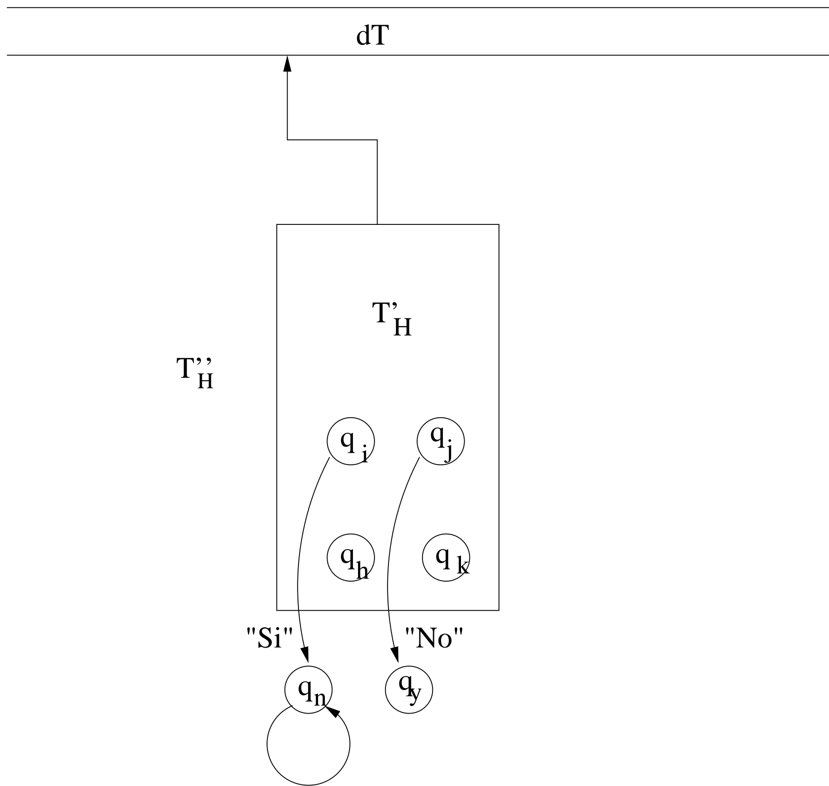


Figura 7.7: La máquina de Turing T''_H.

Capítulo 8

El Problema de la Palabra

Diccionarios como Programas

La idea de buscar una palabra en un diccionario es lo suficientemente común y sencilla para cualquiera, pero lleva directamente a un problema que no puede ser resuelto por una computadora. El problema se basa en la equivalencia de cadenas. Por ejemplo, dada una afirmación, escójase una palabra aleatoriamente, use un diccionario para seleccionar una palabra equivalente, y entonces substitúyase la nueva palabra en la afirmación original. ¿Hasta cuándo son las dos afirmaciones equivalentes bajo una secuencia de estas operaciones?

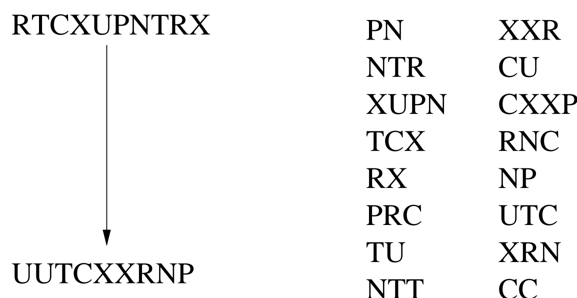
La pregunta parece bastante trivial cuando se imaginan dos afirmaciones. Ciertamente, se sabe cuándo dos afirmaciones son equivalentes: sólo requieren tener el mismo significado. Pero una computadora no cuenta con esa ventaja. Hasta ahora, no hay un programa que entienda de significados.

Parece ser más fácil apreciar el problema computacionalmente si se considera otro lenguaje desconocido, como el griego antiguo. Supóngase que se cuenta con dos fragmentos de escritura de la Grecia antigua: un texto con dos afirmaciones y una parte de un diccionario de griego antiguo. Se cuenta con lo suficiente del diccionario para establecer si las afirmaciones son equivalentes. Las afirmaciones escritas en el texto griego no cuentan con espacios. Si se toman estas dos cadenas de caracteres, ¿serán equivalentes? Para averiguar esto, sistemáticamente se hacen substituciones de palabras en la primera de las afirmaciones, con la esperanza de obtener tras algunos cambios la segunda. Por supuesto, sólo se tienen las substituciones disponi-

bles en el diccionario. Si se tiene una palabra, ésta puede substituirse por su "significado". A primera vista, este procedimiento parece lo suficientemente sólido y directo, pero tiene una falla. De hecho, no hay un procedimiento que funcione.

En 1914, Axel Thue, un matemático noruego, hizo la primera enunciación formal de este problema: Sea Σ un alfabeto finito de símbolos arbitrarios, y sea D un diccionario consistente de un número finito de pares (X_i, Y_i) de palabras. Dada una palabra arbitraria X sobre el alfabeto Σ, una substitución involucra encontrar una sub-palabra X_i en X y substituir su correspondiente palabra Y_i. Las palabras X y Y son equivalentes si hay una secuencia finita de substituciones llevando de X a Y.

Para un ejemplo más humilde (y algo más legible) de este problema, se puede utilizar el alfabeto latino:



¿Es posible convertir la palabra superior izquierda a la palabra inferior izquierda mediante substitución de secuencias del diccionario? El ejecicio puede ser extenuante para un lector humano, pero seguramente una computadora podría ser programada para determinar la equivalencia de estas palabras. El método implícito en el algoritmo descrito anteriormente podría comenzar con la palabra superior izquierda y derivar todas las posibles substituciones en ella (figura 8.1).

Manteniendo una lista de todas las palabras que se han derivado, el programa simplemente itera este mismo procedimiento para cada palabra en la lista. Las palabras en las que se hace una substitución se reemplazan en la lista por las palabras resultantes. Como cada nueva palabra se añade a la lista, la computadora intenta hacerla coincidir con UUTCXXRNP. Si esta palabra es equivalente a RTCXUPNTRX, el programa encontrará la coincidencia tarde o temprano.

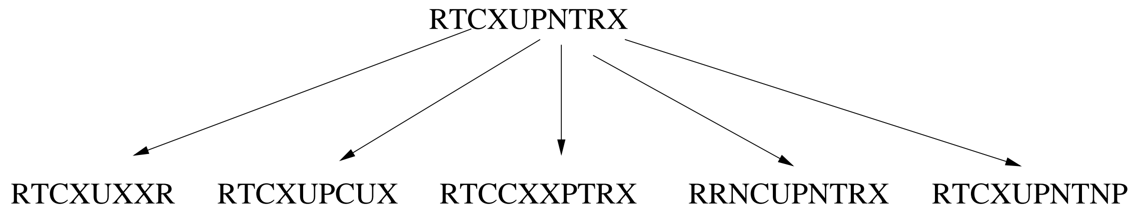


Figura 8.1: Comienzo del proceso de generación de palabras.

Este último enunciado hace pensar que el problema ha sido resuelto, pero no es así. Aun si las dos palabras fueran equivalentes, la secuencia de substituciones volviendo una palabra a la otra puede involucrar palabras intermedias que son arbitrariamente largas. No hay, consecuentemente, una forma de saber cuándo terminará el cómputo. Si las dos palabras no son equivalentes, la computadora podría continuar hasta el infinito, mientras se espera pacientemente la respuesta.

Una de las caraterísticas cardinales de la computabilidad es que el cómputo debe detenerse tarde o temprano en todos los casos. Para ser computable, una función (aun aquélla que involucre una simple respuesta de "sí" o "no") debe ser computable en tiempo finito.

La falla del algoritmo de búsqueda propuesto no comprueba que el problema de la palabra (también conocido como el problema de la palabra para semigrupos) no tiene una solución computacional. Después de todo, ¿cómo se sabe que no existe una sutil teoría que puede ser implementada en un algoritmo totalmente diferente? Quizá tal algoritmo puede hasta ejecutarse en tiempo lineal, es decir, puede determinar la equivalencia de dos cadenas de longitud n en $O(n)$ pasos.

El hecho de que no existe algoritmo alguno para este problema se puede observar más claramente mediante utilizar un truco matemático muy antiguo: desarrollar una transformación entre el problema a la mano y algún otro del cual se sabe algo más. La transformación se conoce como una "reducción de Turing" (*Turing reduction*).

Ya que la dificultad encontrada en la solución propuesta para este problema se refiere a la detención, es posible que el problema de la palabra esté relacionado con el problema de la detención para máquinas de Turing. Esta suposición resulta ser correcta. Para ver cómo, debe ser posible conver-

tir una máquina de Turing a un problema de la palabra. Primero, se enuncian ambos problemas explícitamente. Respectivamente, ambos requieren encontrar algoritmos que hagan lo siguiente: dada una máquina de Turing arbitraria y una cinta inicial, el primer algoritmo determina si la máquina se detiene. Ahora bien, dado un diccionario y pares de palabras, el segundo algoritmo determina si las palabras son equivalentes. El paralelismo en ambas descripciones lleva a preguntarse si la reducción involucra reemplazar una máquina de Turing por alguna clase de diccionario. La noción clave que lleva a tal reemplazo es la descripción instantánea de una máquina de Turing y el ambiente de su cinta. Meramente copia toda la porción no vacía o en blanco de la cinta de la máquina de Turing. Entonces, a la izquierda de la celda actual, insértese una nueva celda que contenga un nombre simbólico para el estado actual de la máquina. Por ejemplo, si la cinta de la máquina de Turing actualmente se viera como:

	0	1	1	#	0	#	0	0	0	1	#	0	1	1	0	

Supóngase que la máquina se encuentra en la celda número 10 contando desde la izquierda, y se encuentra en el estado 5. Reemplácese el 5 por algún símbolo que no se encuentre en el alfabeto de la cinta, por ejemplo, F. Ahora, la cinta queda así:

	0	1	1	#	0	#	0	0	0	F	1	#	0	1	1	0

Parece claro ahora cómo proceder. Constrúyase un diccionario que reproduzca el efecto local en la cinta del programa de la máquina de Turing cuando se le aplique a la posición de la cabeza lectora/escritora.

Si por ejemplo cuando la máquina está en el estado 5 y lee un 1, entra al estado 2 y escribe un 0, moviéndose una celda a la izquierda, el diccionario se compodría de las siguientes entradas:

0F1	F00
1F1	F10
#F1	F#0

En este caso, se aplicaría la primera entrada: reemplácese los contenidos de la celda 0F1 por F00.

Este proceso, sin embargo, no es más que un ejemplo de las substituciones utilizadas en el problema de Thue. La reducción completa, desafortunadamente, involucra algo más. Dos palabras deben proveerse para finalmente llegar adecuadamente a una instancia del problema de la palabra. Sea la primera palabra el estado inicial de la cinta, con el marcador de estado ya considerado a la izquierda de la celda inicial.

Podría parecer sobre-restrictivo, pero es perfectamente correcto permitir que la segunda palabra este compuesta enteramente de ceros, o tal vez algún símbolo especial, por ejemplo, Z. Cualquier máquina de Turing puede modificarse para producir este efecto, si llega a detenerse.

Claramente, es notorio que el proceso de reducción es Turing-computable. Esto significa simplemente que es computable. Las dos palabras y el diccionario pueden fácilmente ser procesadas por un algoritmo que tome el programa de la máquina de Turing y la cinta inicial como entrada.

Se sabe que no hay un algoritmo que resuelva el problema de la detención. Se tiene, sin embargo, una transformación computable de este problema al problema de la palabra. Si éste último tuviera solución, también la tendría el problema de la detención, lo que no es posible.

Bibliografía

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] D.I.A. Cohen *Introduction to Computer Theory.* Jophn Wiley, 1986.

[3] D.R. Hofstadter *Gödel, Escher, Bach: An Eternal Golden Braid.* Basic Books, 1979.

[4] S.C. Kleene *Introduction to Metamathematics.* Van Nostrand, 1950.

[5] H.R. Lewis and C.H. Papadimitriou *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[6] D.W. Loveland *Automated Theorem Proving: A Logical Basis.* North Holland, 1978.

[7] D.E. Knuth *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, 1969.

[8] M.L. Minsky *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.