Notas del Curso:
Funciones Recursivas y Máquinas de Turing

Facultad de Ciencias de la Computación
Benemérita Universidad Autónoma de Puebla

M.I.A. José Juan Palacios Pérez
jpalacio@cs.buap.mx
http://www.cs.buap.mx/∼jpalacio

Verano 2002

El propósito inicial de la *teoría de computabilidad* es precisar la idea intuitiva de *función computable*, esto es, una función cuyos valores pueden calcularse automáticamente (o de manera efectiva). Ello conduce, sin embargo, a explorar el concepto de computabilidad de una forma matemática (formal), como también las distintas ideas relacionadas tales como *decidibilidad* y *enumerabilidad efectiva*.

Puede describirse a la teoría de computabilidad, desde el punto de vista de las ciencias de la computacion, mediante las siguientes consideraciones:

1. ¿Qué pueden hacer las computadoras en *principio* (sin restricción de espacio, tiempo ó dinero)?

   - y en consecuencia: -

2. ¿Cuáles son las *limitaciones* teóricas inherentes?

Por tanto, lo que se discutirá en este curso es independiente de las computadoras reales (vistas como dispositivos de símbolos físicos), y de los lenguajes y técnicas de programación. La materia que se discute es parte del transfondo teórico de las computadoras y su uso, de profundo interés para la comunidad de científicos de computación, programadores, analistas, etc.

El objetivo principal de éste curso es que el alumno aprenda la noción intuitiva de lo que puede ser *computable*, i.e., realizado de manera efectiva por una computadora; a su vez, que tal noción de *computabilidad* se puede expresar mediante distintos formalismos. Sin embargo, apreciará la equivalencia de tales formalismos y entenderá los límites de lo que puede ser computable.

De manera específica, se espera cubrir los siguientes objetivos particulares:

1. El estudiante estudiará al modelo de cómputo abstracto conocido como *máquina de registros ilimitados* (URM) y a las *funciones recursivas parciales*. Tales modelos permiten expresar la noción de *computable mediante un algoritmo*.

2. El alumno aprenderá lo que significan los conceptos *decidibilidad*, *indecidibilidad* y *decidibilidad parcial*.

3. El alumno conocerá de qué manera la idea de función computable se conecta con la lógica matemática.

4. El alumno aprenderá que lo computable describe la misma clase de funciones, independientemente del enfoque para formalizar el concepto de computabilidad.

5. El alumno manejará las técnicas de reducción entre problemas y los grados de indecidibilidad.

El propósito de éstas notas es servir como referencia básica para el curso. Están fundamentadas principalmente en el texto de Nigel Cutlan: *Computability, an introduction to recursive function theory*; pudiéndo considerarse esencialmente como una antología de tal texto.

A la dulce memoria de
ESTELA ANDRADE ESCOBAR
1973 - 2002
brillará por siempre tu luz en mi corazón.

Contenido

Capítulo 1

Funciones Computables

Comenzamos discutiendo la idea fundamental de un algoritmo ó procedimiento efectivo. A continuación se describirá la forma en la cual ésta idea puede llevarse a cabo de manera precisa utilizando un tipo de computadora ideal, ésto forma parte de los fundamentos para la teoría matemática de computabilidad y funciones computables. Éste capítulo asume familiaridad con la teoría de conjuntos, relaciones y funciones, funciones definidas sobre los números naturales[1] y notación de la lógica matemática clásica (cálculo de predicados de primer orden con igualdad).

1.1 Algoritmos ó procedimientos efectivos

En aritmética se aprende, entre otras cosas, a sumar y multiplicar dos números mediante métodos o reglas. Tales métodos son ejemplos de *algoritmos ó procedimientos efectivos*. Su implantación no requiere "inteligencia" en el sentido que tales métodos describen una serie de pasos ó instrucciones elementales y precisas.

De manera general, un *algoritmo* es una regla mecánica, ó método automático, ó programa para realizar una operación matemática. Algunos ejemplos de operaciones en las cuales se pueden describir algoritmos fácilmente son

1. dado un número entero n, encontrar el n-ésimo número primo,
2. derivar un polinomio,
3. encontrar el mayor factor común de dos números (algoritmo de Euclides),
4. dados dos números x, y decidir si x es múltiplo de y.

Los algoritmos se pueden representar informalmente mediante una *caja negra* que recibe una *entrada* sobre la cual se realizará la operación y ofrece una *salida ó resultado* de la operación. Tal salida se produce mecánicamente por la

[1]Denotaremos por $\mathcal{N}$ al conjunto de los naturales, incluyendo el 0.

caja negra (una calculadora, computadora, un perro bien entrenado, etc.). El algoritmo es el procedimiento o método que se llevó a cabo por la caja negra para obtener la salida a partir de la entrada.

$$\text{Entrada} \rightarrow \boxed{\text{Caja Negra}} \rightarrow \text{Salida}$$

Cuando un algoritmo se utiliza para calcular los valores de una función numérica entonces la función en cuestión se describe mediante frases como *efectivamente calculable* o *algorítmicamente computable* o *efectivamente computable* ó solamente *computable*. Por ejemplo, las funciones xy, $HCF(x, y)$ =mayor factor común de x y y, y $f(n)$ = el n-ésimo número primo, son todas computables en éste sentido informal.

Por otra parte, considérese a la función:

$$g(n) = \begin{cases} 1 & \text{si existe una serie de exactamente } n \text{ 7's} \\ & \text{consecutivos en la expansión decimal de } \pi, \\ 0 & \text{en otro caso} \end{cases}$$

Podría aceptarse que g es una función legítima, pero ¿es computable? Existe un procedimiento mecánico para generar dígitos sucesivos en la expansión decimal de π, de tal suerte que se puede sugerir el siguiente algoritmo:

'Dado n, comenzar generando la expansión decimal de π un dígito a la vez y ver si se trata del 7. Si en alguna etapa ocurren exactamente n 7's entonces detenerse y regresar $g(n) = 1$. Si no existe tal secuencia de 7's entonces poner $g(n) = 0$.'

El problema con éste 'procedimiento' es que, si para un n en particular no hay una secuencia de exactamente n 7's consecutivos entonces no hay etapa en el proceso donde nos podamos detener y concluír. La secuencia de n 7's consecutivos, hasta donde podemos saber, podría ocurrir en la expansión de π que aún no hemos examinado; de tal suerte que éste 'procedimiento' podría en principio continuar por siempre para entradas n tales que $g(n) = 0$; por tanto, no se trata de un procedimiento *efectivo*.

Este ejemplo ilustra dos aspectos implícitos en la idea de un procedimiento efectivo:

1. tal procedimiento se realiza en una secuencia de pasos ó etapas (cada una de las cuales se completa en un tiempo finito),

2. cualquier salida ó resultado emerge tras un número finito de pasos.

Éstas ideas de algoritmo (ó procedimiento efectivo) y la noción asociada de función computable, se precisarán antes de que sean constituídas como las bases de una teoría matemática de la computabilidad.

$$\boxed{\text{Funciones Computables}}$$
$$\updownarrow$$
$$\text{Algoritmo}$$

Los procedimientos que pueden realizarse mediante computadoras son ejemplos de procedimientos efectivos. No obstante, cualquier computadora real independiente está limitada tanto en el espacio de trabajo (almacenamiento) y en el tamaño de la entrada que pueda recibir; por tanto, construiremos las definiciones formales de algoritmo y función computable en términos de un modelo de *computadora ideal simple*. Esta computadora es ideal en el sentido que sus programas son finitos y un cálculo toma a su vez un número finito de pasos; sin embargo no tiene una limitación en cuanto al tamaño de la entrada ni en cuanto al espacio de almacenamiento. Tanto las entradas como las salidas estarán restringidas a valores de números naturales, lo cual de ninguna manera representa una restricción importante dado que las operaciones que involucran otros tipos de objetos pueden codificarse como operaciones en los naturales.

1.2 La máquina de registros ilimitados

El modelo de computadora ideal que utilizaremos se llama *máquina de registros ilimitados*[2], la cual es una ligera variante de la máquina concebida por Sheperson & Sturgis en 1963.

La URM está compuesta por un número infinito de *registros* etiquetados por $R_1, R_2, ...$, cada uno de los cuales en cualquier instante contiene un *número natural*; denotaremos al número contenido en R_n por r_n, lo cual se puede representar de la siguiente manera

R_1	R_2	R_3	R_4	R_5	R_6	R_7	...
r_1	r_2	r_3	r_4	r_5	r_6	r_7	...

El contenido de los registros puede alterarse por la URM en respuesta a ciertas *instrucciones* que la máquina reconoce. Tales instrucciones corresponden a operaciones muy simples utilizadas en la realización de cálculos con números. Una lista finita enumerada de instrucciones constituye un *programa P*. Las instrucciones son de cuatro tipos y las definimos operacionalmente:

Cero Para cada $n = 1, 2, 3, ...$ existe una instrucción cero $Z(n)$ cuyo efecto es cambiar el contenido del registro R_n a 0, sin alterar el contenido de los demás registros

Supóngase que la URM está en la siguiente configuración

R_1	R_2	R_3	R_4	R_5	R_6	...
9	6	5	23	7	0	...

y se procede a aplicar la instrucción $Z(3)$, lo cual produce la siguiente configuración:

R_1	R_2	R_3	R_4	R_5	R_6	...
9	6	0	23	7	0	...

[2] *Unlimited Register Machine (URM).*

Denotamos la respuesta de la URM a una instrucción cero $Z(n)$ como $0 \to R_n$, o también $r_n := 0$.

Sucesor Para cada $n = 1, 2, 3, ...$ existe una instrucción sucesor $S(n)$ cuyo efecto es incrementar el valor contenido en R_n en 1, dejando a los demás registros inalterados.

Si aplicamos la instrucción $S(5)$ a la URM en la configuración anterior obtendremos la nueva configuración

R_1	R_2	R_3	R_4	R_5	R_6	...
9	6	0	23	8	0	...

Denotaremos el efecto de una instrucción sucesor $S(n)$ mediante $r_n + 1 \to R_n$, ó $r_n := r_n + 1$.

Transferencia Para cada $n = 1, 2, 3, ...$ y cada $m = 1, 2, 3, ...$ existe una instrucción transferencia $T(m, n)$ cuyo efecto es reemplazar el contenido de R_n por el contenido de R_m (i.e., transferir r_m a R_n), los demás registros (incluyendo R_m) quedan inalterados.

Para la URM en la configuración anterior, aplicamos la instrucción $T(5, 1)$ y tiene el siguiente efecto

R_1	R_2	R_3	R_4	R_5	R_6	...
8	6	0	23	8	0	...

La respuesta de la URM a la instrucción $T(m, n)$ se denota por $r_m \to R_n$ ó $r_n := r_m$.

Salto en la ejecución de un algoritmo informal podría suceder una etapa en la cual se describan cursos alternativos de acción dpendiendo del progreso en la operación hasta esa etapa. En otras situaciones podría ser necesario repetir una rutina dada varias veces. La URM puede reflejar tales procedimientos mediante el uso de las instrucciones de salto, las cuales permiten saltar hacia delante ó hacia atrás en la lista de instrucciones.

Para cada $n = 1, 2, 3, ...$ y cada $m = 1, 2, 3, ...$ y $q = 1, 2, 3, ...$ existe una instrucción de salto $J(m, n, q)$, cuyo efecto es el siguiente: supóngase que la instrucción actual es la i-ésima en P ($i \leq s$, donde el programa tiene s instrucciones), entonces se comparan los contenidos de R_m y R_n y se realiza lo siguiente:

- si $r_m = r_n$ la URM procede a ejecutar la instrucción q de P;
- si $r_m \neq r_n$ la URM ejecuta la siguiente instrucción (i.e., la $i + 1$).

Si el salto no es posible debido a que P tiene menos de q instrucciones entonces la URM detiene su operación.

Tipo	Instrucción	Respuesta de la URM
Cero	$Z(n)$	Reemplazar r_n por 0. ($0 \to R_n$, $r_n := 0$)
Sucesor	$S(n)$	Sumar 1 a r_n ($r_n + 1 \to R_n$, ó $r_n := r_n + 1$)
Transferencia	$T(m, n)$	Reemplazar r_n por r_m ($r_m \to R_n$ ó $r_n := r_m$)
Salto	$J(m, n, q)$	Si $r_m = r_n$ ejecutar la instrucción q; de lo contrario, ejecutar la siguiente instrucción.

Figura 1.1: Instrucciones de la URM.

La Figura 1.1 sintetiza la respuesta de la URM a las cuantro instrucciones anteriores. Las instrucciones cero, sucesor y transferencia se denominan instrucciones *aritméticas.*

Para realizar un **Cómputo** la URM necesita un programa P y una *configuración inicial* i.e., una secuencia de valores $a_1, a_2, ...$ colocados en los registros $R_1, R_2, ...$ Supóngase que P contiene s instrucciones $I_1, I_2, ..., I_s$. La URM comienza el cómputo obedeciendo a I_1 y continúa ejecutando las instrucciones siguientes. Para cualquier etapa futura del cómputo, supóngase que la URM está obedeciendo la instrucción I_k, entonces, habiéndola finalizado, procede a ejecutar *la siguiente instrucción* en el cómputo bajo el siguiente criterio:

- si I_k no es una instrucción de salto, entonces la siguiente instrucción es I_{k+1};
- si $I_k = J(m, n, q)$ entonces la siguiente instrucción es

$$\begin{cases} I_q & \text{si } r_m = r_n \\ I_{k+1} & \text{de lo contrario.} \end{cases}$$

La URM procede tanto como le sea posible; el cómputo *se detiene* cuando y sólo cuando, no exista instrucción siguiente; i.e., si la URM solamente ha obedecido a la instrucción I_k y la siguiente instrucción bajo el criterio mencionado anteriormente es I_v donde $v > s$. Esto puede suceder en los siguientes casos:

1. si $k = s$ (la última instrucción en P se ha obedecido) e I_s es una instrucción aritmética,
2. si $I_k = J(m, n, q)$, $r_m = r_n$ y $q > s$,
3. si $I_k = J(m, n, q)$, $r_m \neq r_n$ y $k = s$.

Decimos que el cómputo se detiene después de la instrucción I_k; la *configuración final* es la secuencia de los contenidos de los registros en ésta etapa.

Ejemplo 1 *Considérese el siguiente programa:*

$$\begin{array}{ll|ll} I_1 & J(1,2,6) & I_4 & J(1,2,6) \\ I_2 & S(2) & I_5 & J(1,1,2) \\ I_3 & S(3) & I_6 & T(3,1) \end{array}$$

Sea la configuración inicial de la URM:

R_1	R_2	R_3	R_4	R_5	...
9	7	0	0	0	...

Representamos el progreso del cómputo mostrando las configuraciones sucesivas en la cual se encuentra la máquina junto con la instrucción que será aplicada en cada etapa:

R_1	R_2	R_3	R_4	R_5	...	*Sig. instrucción*
9	7	0	0	0	...	I_1
9	7	0	0	0	...	I_2 *(ya que* $r_1 \neq r_2$*)*
9	8	0	0	0	...	I_3
9	8	1	0	0	...	I_4
9	8	1	0	0	...	I_5 *(ya que* $r_1 \neq r_2$*)*
9	8	1	0	0	...	I_2 *(ya que* $r_1 = r_1$*)*
9	9	1	0	0	...	I_3
9	9	2	0	0	...	I_4
9	9	2	0	0	...	I_6 *(ya que* $r_1 = r_1$*)*
9	9	2	0	0	...	I_7:STOP

Éste cómputo se detiene dado que no hay instrucción 7 en el programa P.

Sin embargo, existen cómputos que nunca se detinen: por ejemplo, ningún cómputo bajo el programa $S(1), J(1,1,1)$ se detiene, el cual describe un *ciclo infinito.*

Ejercicios 1 *Muestre que el cómputo bajo el programa del ejemplo 1 con la configuración inicial* $2, 3, 0, 0, 0, ...$ *nunca se detiene.*

Existen formas más sofisticadas en las cuales un cómputo puede realizarse por siempre, pero esencialmente se trata del tipo de ciclo infinito que acaba de ser descrito. La pregunta de *decidir* cuándo un cómputo en particular se detiene o no es un problema esencial que será discutido posteriormente.

Utilizaremos la siguiente notación para nuestras discusiones siguientes:

Sea $a_1, a_2, ...$ una secuencia infinita de valores $\mathcal{N}$ y sea P un programa. Denotamos

1. $P(a_1, a_2, ...)$ el cómputo bajo P con la configuración inicial $a_1, a_2, ...$;
2. $P(a_1, a_2, ...) \downarrow$ para indicar que el cómputo $P(a_1, a_2, ...)$ se detiene.
3. $P(a_1, a_2, ...) \uparrow$ para indicar que el cómputo $P(a_1, a_2, ...)$ nunca se detiene.

   Usaremos la convención siguiente para indicar cuándo una secuencia infinita esté conformada por una subsecuencia finita $a_1, a_2, ..., a_n$ y el resto sean todos 0:
4. $P(a_1, a_2, ..., a_n)$ denota el cómputo $P(a_1, a_2, ..., a_n, 0, 0, ...)$.
5. $P(a_1, a_2, ..., a_n) \downarrow$ denota $P(a_1, a_2, ..., a_n, 0, 0, ...) \downarrow$.

6. $P(a_1, a_2, ..., a_n) \uparrow$ denota $P(a_1, a_2, ..., a_n, 0, 0, ...) \uparrow$.

Un cómputo que termina se dice que *converge* y uno que nunca termina se dice que *diverge*.

1.3 Funciones URM computables

¿Qué significa que una función f definida en los naturales $f : \mathcal{N}^n \to \mathcal{N}$ $(n \geq 1)$ sea computable por la URM? Intuitivamente, podemos considerar **calcular** el valor $f(a_1, a_2, ..., a_n)$ **mediante un programa** P teniendo como configuración inicial $a_1, a_2, ..., a_n, 0, 0, ...$ (de acuerdo a la notación expuesta, $P(a_1, a_2, ..., a_n)$). Si tal cómputo[3] se detiene, necesitamos tener un único valor para considerarlo como resultado ó salida del cómputo; por convención, consideraremos que al final del cómputo tal número estará colocado en R_1.

Dado que un cómputo podría no detenerse ($P(a_1, a_2, ..., a_n) \uparrow$) permitiremos que la definición de computabilidad se aplique a las funciones parciales $f : \mathcal{N}^n \to \mathcal{N}$ (i.e., cuyo dominio no sea $\mathcal{N}^n$ propiamente). Se requerirá que los cómputos relevantes se detengan (y produzcan el resultado correcto!) *precisamente* para entradas que estén dentro del dominio de la función f.

Definición 1 *Sea f una función parcial $f : \mathcal{N}^n \to \mathcal{N}$.*

1. *Sea P un programa y sean $a_1, a_2, ..., a_n, b \in \mathcal{N}$.*

   (a) *El cómputo $P(a_1, a_2, ..., a_n)$* **converge** *a* **b** *si $P(a_1, a_2, ..., a_n) \downarrow$ y en la configuración final b se encuentra en R_1; denotación $P(a_1, a_2, ..., a_n) \downarrow b$.*

   (b) *Se dice que P computa-URM a f si, para toda $a_1, a_2, ..., a_n, b$ $P(a_1, a_2, ..., a_n) \downarrow b$ si y sólo si $(a_1, a_2, ..., a_n) \in dom(f)$ y evidentemente $f(a_1, a_2, ..., a_n) = b$.*

2. *La función f es* **URM-computable** *si existe un programa que computa-URM a f.*

Denotaremos por $\mathcal{C}$ a la clase de funciones URM-computables y mediante $\mathcal{C}_n$ a la clase de funciones n-arias URM-computables. Consideraremos indistintamente los términos *computable* y *URM- computable*, a menos que lo indiquemos de otra manera explícitamente.

Ejemplo 2 *Sea la función $f(x, y) = x + y$.*

Se puede evaluar el resultado mediante y incrementos a x. El programa correspondiente comienza con la configuración inicial $x, y, 0, 0, ...$ y utilizará al registro R_3 como contador para saber cuántas veces r_1 ha sido incrementado. La configuración típica del cómputo es

R_1	R_2	R_3	R_4	R_5	...
$x+k$	y	k	0	0	...

[3] En general pueden existir varios cómputos.

El programa se detendrá cuando $k = y$ dejando el resultado $x + y$ en R_1 como se requiere:

$$\begin{array}{ll|ll} I_1 & J(3,2,5) & I_3 & S(3) \\ I_2 & S(1) & I_4 & J(1,1,1) \end{array}$$

Por tanto, $f(x,y) = x + y$ es computable.

Ejemplo 3 *Sea* $x \dot{-} 1 = \begin{cases} x - 1 & si\ x > 0, \\ 0 & si\ x = 0. \end{cases}$

Dada la configuración inicial $x, 0, 0, ...$ se verificará primero si $x = 0$, de ser el caso, el cómputo finaliza. De lo contrario mantener dos contadores, uno tendrá k y el otro $k + 1$, comenzando con $k = 0$. La configuración típica del cómputo es

R_1	R_2	R_3	R_4	...
x	k	$k+1$	0	...

El programa finalizará cuando $x = k + 1$ y entonces el resultado es k; de lo contrario se incrementan ambos contadores y volver a revisar:

$$\begin{array}{ll|ll} I_1 & J(1,4,9) & I_5 & S(3) \\ I_2 & S(3) & I_6 & J(1,1,3) \\ I_3 & J(1,3,7) & I_7 & T(2,1) \\ I_4 & S(2) & & \end{array}$$

Por tanto la función $x \dot{-} y$ es computable.

Ejemplo 4 *Sea* $f(x) = \begin{cases} \frac{1}{2}x & si\ x\ es\ par, \\ indefinido & si\ x\ es\ impar. \end{cases}$

Para éste ejemplo, $dom(f) =$E (el conjunto de los naturales pares), de tal forma que se evitará que el programa se detenga ante una entrada impar. Se tienen dos contadores que contienen k y $2k$ para $k = 0, 1, 2, ...$; para valores sucesivos de k se debe verificar cuando $x = 2k$ en cuyo caso la respuesta es k; de otra manera incrementar k en 1 y repetir. Si x es impar, el programa continuará por siempre. La configuración típica para éste cómputo es

R_1	R_2	R_3	R_4	...
x	$2k$	k	0	...

inicialmente, $k = 0$. El programa correspondiente es

$$\begin{array}{ll|ll} I_1 & J(1,2,6) & I_4 & S(2) \\ I_2 & S(3) & I_5 & J(1,1,1) \\ I_3 & S(2) & I_6 & T(3,1) \end{array}$$

Por tanto la función $f(x)$ es computable.

Dado cualquier programa P y $n \geq 1$ veremos que existe una única función n-aria f que P computa bajo la configuración inicial $a_1, a_2, ..., a_n, 0, 0, ...$, la cual denotaremos por $f_P^{(n)}$

$$f_P^{(n)}(a_1, a_2, ..., a_n) = \begin{cases} \textbf{b} \text{ único, tal que } P(a_1, a_2, ..., a_n) \downarrow b, \\ \quad \text{si } P(a_1, a_2, ..., a_n) \downarrow \\ \text{indefinido si } P(a_1, a_2, ..., a_n) \uparrow \end{cases}$$

Más adelante se considerará el problema de determinar a $f_P^{(n)}$ para cualquier programa P.

En general, una función computable puede ser computada por muchos programas distintos (en cualquier programa pueden insertarse instrucciones que no tengan efecto). Sin embargo, puede ocurrir que programas distintos P_1, P_2 calculen la misma función, i.e., $f_{P_1}^{(n)} = f_{P_2}^{(n)}$ para algún (o toda) n. Más adelante se considerará el problema de decidir cuando dos o más programas computan la misma función.

Ejercicios 2 *1. Mostrar que las siguientes funciones son computables construyendo programas URM que las computen.*

(a) $f(x) = \begin{cases} 0 & si\ x = 0, \\ 1 & si\ x \neq 0. \end{cases}$

(b) $f(x) = 5$

(c) $f(x, y) = \begin{cases} 0 & si\ x = y, \\ 1 & si\ x \neq y. \end{cases}$

(d) $f(x, y) = \begin{cases} 0 & si\ x \leq y, \\ 1 & si\ x > y. \end{cases}$

(e) $f(x) = \begin{cases} \frac{1}{3}x & si\ x\ es\ múltiplo\ de\ 3, \\ indefinido & en\ otro\ caso. \end{cases}$

(f) $f(x) = [2x/3]$*, donde* $[z]$ *denota el mayor entero* $\leq z$*.*

(g) Para el programa P *del ejemplo 1, ¿quién es* $f_P^{(2)}$*?*

(h) Supónga que P *es un programa sin ninguna instrucción de salto, muestre que existe un número* m *tal que ya sea*

$$f_P^{(1)}(x) = m, \quad para\ toda\ x$$

ó

$$f_P^{(1)}(x) = x + m, \quad para\ toda\ x$$

(i) Muestre que para cada instrucción de transferencia $T(m, n)$ *existe un programa sin instrucciones de transferencia que tiene el mismo efecto que* $T(m, n)$ *bajo cualquier configuración de la URM.*

1.4 Predicados y problemas decidibles

En matemáticas, una tarea bastante común es *decidir* cuándo los números poseen alguna propiedad. Por ejemplo, si dados dos números x, y decidir si x tiene la propiedad de ser múltiplo de y. Un algoritmo para realizar ésta operación será un procedimiento efectivo que ante las entradas x, y responderá como salida *'Sí'* ó *'No'*. Adoptamos la convención de que 1 significa *'Sí'* y 0 significa *'No'*, de tal forma que la función puede expresarse como

$$f(x,y) = \begin{cases} 1 & \text{si } x \text{ es múltiplo de } y, \\ 0 & \text{si } x \text{ no es múltiplo de } y. \end{cases}$$

Podemos afirmar que la propiedad ó *predicado 'x es múltiplo de y'* es *algorítmica* ó *efectivamente decidible*, o simplemente *decidible* si tal función f es computable.

Generalizaremos esta idea mediante las siguientes definiciones.

Definición 2 *Supóngase que $M(x_1, x_2, ..., x_n)$ es un predicado n-ario definido sobre $\mathcal{N}$. La* **función característica** *de M, denotada por $c_M(\vec{x})$ (para $\vec{x} = (x_1, x_2, ..., x_n)$) está definida como*

$$c_M(\vec{x}) = \begin{cases} 1 & \textit{si } M(x) \textit{ se cumple (es Verdadero)}, \\ 0 & \textit{si } M(x) \textit{ no se cumple.} \end{cases}$$

Definición 3 *El predicado $M(\vec{x})$ es* **decidible** *si su función c_M es computable; $M(\vec{x})$ es* **indecidible** *si $M(\vec{x})$ no es* **decidible**.

Ejemplo 5 *El siguiente predicado es decidible:*

1. *'$x = 0$', su función característica está dada por*
   $$g(x) = \begin{cases} 1 & \textit{si } x = 0, \\ 0 & \textit{si } x \neq 0. \end{cases}$$
   El siguiente programa computa a g: $\begin{array}{ll|ll} I_1 & J(1,2,3) & I_3 & S(2) \\ I_2 & J(1,1,4) & I_4 & T(2,1) \end{array}$

En el contexto de la decidibilidad, las propiedades o predicados a menudo se denominan *problemas*. Nótese que cuando se discute respecto a la decidibilidad (e indecidibilidad) siempre es de interés la computabilidad (o no computabilidad, respectivamente) de **funciones totales**.

Ejercicios 3 *Muestre que los siguientes predicados son decidibles:*
'$x < y$', '$x \neq 3$', 'x es par'.

1.5 Computabilidad en otros Dominios

Dado que la URM únicamente utiliza números naturales, la definición de computabilidad e indecidibilidad se aplica solamente a las funciones y predicados definidos en $\mathcal{N}$. Tales nociones se pueden extender naturalmente a otros tipos de objetos (e.g. enteros, polinomios, matrices, etc.) mediante *codificación*, de la siguiente manera.

Definición 4 *Una* **codificación** *de un dominio D de objetos es una función inyectiva efectiva (y explícita) $\alpha : D \to \mathcal{N}$. Decimos que un objeto $d \in D$ es* codificado *por un número natural $\alpha(d)$.*

Supóngase que f es una función definida $f : D \to D$, f puede codificarse mediante la función $f^* : \mathcal{N} \to \mathcal{N}$ que establece la correspondencia entre la codificación de un objeto $d \in dom(f)$ hacia $f(d)$. De manera explícita, se tiene

$$f^* = \alpha \circ f \circ \alpha^{-1}$$

Ahora se puede extender la definición de computabilidad URM hacia D indicando que f es computable si f^* es computable en $\mathcal{N}$.

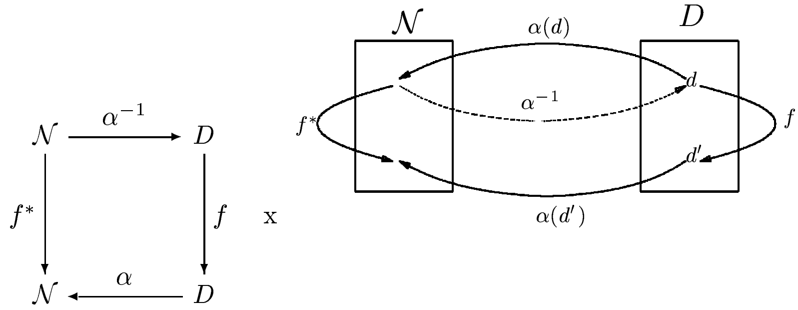

Figura 1.2: Codificación en otros dominios.

Ejemplo 6 *Considérese el dominio de los enteros $\mathcal{Z}$. Una codificación explícita está dada por la función α definida como*

$$\alpha(n) = \begin{cases} 2n & si\ n \geq 0, \\ -2n-1 & si\ n < 0. \end{cases}$$

A su vez, α^{-1} está dada por

$$\alpha^{-1}(m) = \begin{cases} \frac{1}{2}m & si\ m\ es\ par, \\ -\frac{1}{2}(m+1) & si\ m\ es\ impar. \end{cases}$$

Considérese a la función $f(x) = x - 1$ definida en $\mathcal{Z}$, entonces su correspondiente $f^* : \mathcal{N} \to \mathcal{N}$ está definida por

$$f^*(x) = \begin{cases} 1 & si\ x = 0\ (i.e.,\ x = \alpha(0)\ ), \\ x-2 & si\ x > 0\ y\ x\ es\ par\ (i.e., x = \alpha(n), n > 0\ ), \\ x+2 & si\ x\ es\ impar\ (i.e.,\ x = \alpha(n), n < 0\ ). \end{cases}$$

Se deja como ejercicio al lector escribir un programa que compute f^* y que por tanto demuestra que f es computable en $\mathcal{Z}$.

Las definiciones para funciones computables n-arias y predicados decidibles en D se obtienen extendiendo naturalmente la idea anterior.

Ejercicios 4 *1. Muestre que la función $2x$ definida en $\mathcal{Z}$ es computable.*

2. Muestre que el predicado '$x \geq 0$' es decidible en $\mathcal{Z}$.

Capítulo 2

Generando Funciones Computables

En éste capítulo veremos varios métodos para combinar funciones computables, generando a su vez otras funciones computables. Esto nos permitirá mostrar rápidamente (i.e., sin necesidad de construír un programa correspondiente en cada caso) que muchas funciones bastante conocidas son computables. Por otra parte, implícitamente se introducen *operadores* sobre funciones (o en otras palabras, funciones que toman a su vez a otras funciones).

2.1 Funciones primitivas

A partir de un conjunto de funciones básicas que se definen a continuación se pueden construír funciones computables más complicadas utilizando las técnicas que serán descritas en secciones siguientes.

Lema 1 *Las siguientes funciones básicas son computables:*

1. *la fución* **cero**: $\mathbf{0}(x) = 0, x \in \mathcal{N}$,
2. *la función* **sucesor**: $x + 1, x \in \mathcal{N}$,
3. *la función* **proyección**: U_i^n *dada por* $U_i^n(x_1, x_2, ..., x_n) = x_i$, *para cada* $n \geq 1$ *y* $1 \leq i \leq n$.

Demostración: las funciones anteriores corresponden a las instrucciones aritméticas para la URM; específicamente se tienen los siguientes programas

Función	Programa URM
$\mathbf{0}$	$Z(1)$
$x + 1$	$S(1)$
U_i^n	$T(i, 1)$

$\diamond$

2.2 Concatenando programas

Será necesario en las secciones siguientes construír programas que a su vez incorporen a otros programas como *subprogramas ó subrutinas*, por tanto en ésta sección se definirá cómo realizar tal construcción de manera sencilla.

Como ejemplo, supóngase que tenemos a los programas P y Q y deseamos componerlos de forma tal que P se ejecute primero y entonces Q. Intuitivamente, podría considerarse simplemente colocar las instrucciones de Q inmediatamente enseguida de las instrucciones de P, sin embargo existen dos aspectos importantes a considerar.

2.2.1 Programas en forma estándar

Supóngase que $P = I_1, I_2, ..., I_s$. Un cómputo bajo P se termina cuando *'la siguiente instrucción para el cómputo'* es I_v para $v > s$; requerimos entonces que el cómputo para el programa compuesto proceda ahora con la *primer* instrucción de Q, lo cual sucederá de forma automática si se cumple que $v = s + 1$, pero no en otra forma. Por ello, para concatenar programas preferiremos restringir tal operación de concatenación sobre aquellos programas que se detienen invariantemente gracias a que la siguiente instrucción es I_{s+1}. Tales programas se dice que están en *forma estándar*; sólamente las instrucciones de salto pueden afectar que un programa no esté en forma estándar.

Definición 5 *Un programa $P = I_1, I_2, ..., I_s$ está en forma estándar si, para cada instrucción de salto $J(m, n, q)$ en P se tiene que $q \leq s + 1$.*

(El lector puede revisar si alguno de los programas vistos anteriormente están o no en forma estándar).

Lema 2 *Para todo programa P existe un programa P^* en forma estándar tal que cualquier cómputo bajo P^* es idéntico al correspondiente cómputo bajo P, excepto posiblemente en la manera de detenerse. En particular, para cualquier $a_1, a_2, ..., a_n, b$*

$$P(a_1, a_2, ..., a_n) \downarrow b \text{ si y sólo si } P^*(a_1, a_2, ..., a_n) \downarrow b,$$

y por tanto $f_P^{(n)} = f_{P^*}^{(n)}$ para todo $n > 0$.

Demostración: Sea $P = I_1, I_2, ..., I_s$, para obtener P^* simplemente se cambian las instrucciones de salto de tal forma que todo salto que provoca la finalización del programa ocurre gracias a que la instrucción a saltar será I_{s+1}. Explícitamente, $P^* = I_1^*, I_2^*, ..., I_s^*$ donde

- si I_k no es una instrucción de salto, entonces $I_k^* = I_k$
- si $I_k = J(m, n, q)$ entonces $I_k^* = \begin{cases} I_k & \text{si } q \leq s + 1, \\ J(m, n, s + 1) & \text{si } q > s + 1. \end{cases}$

Claramente, P^* cumple con la forma estándar. $\diamond$

2.2.2 Congruencia en instrucciones de salto

Asúmase que los programas P, Q están en forma estándar y consideramos concatenar P y Q consistentemente. Un salto $J(m, n, q)$ que ocurre en Q debe cambiar el curso de ejecución hacia la instrucción $s + q$ (ya que P tiene s instrucciones); por tanto, cada salto en Q de la forma $J(m, n, q)$ debe modificarse por $J(m, n, s + q)$ en el programa compuesto.

Ahora ya podemos definir la concatenación de programas:

Definición 6 *Sean P y Q programas en forma estándar de longitud s y t respectivamente. La* **concatenación** *de P y Q se denota PQ y es el programa $I_1, I_2, ..., I_s, I_{s+1}, ..., I_{s+t}$ donde $P = I_1, I_2, ..., I_s$ y las instrucciones restantes corresponden a Q en las cuales cada salto $J(m, n, q)$ ha sido reemplazado por $J(m, n, s + q)$.*

Gracias a la definición anterior, cualquier cómputo de PQ es idéntico a los cómputos correspondientes bajo P seguido por Q, cuya configuración inicial para Q es la configuración final del cómputo bajo P.

2.2.3 Subrutinas

Cuando deseamos construír un programa Q que tenga como subrutina un programa dado P debemos cuidar que algunos registros (en el transcurso del cómputo actual) no sean alterados cuando se invoca a P. Dado que P es finito, existe un número más pequeño u tal que ninguno de los registros R_v (para $v > u$) es utilizado en P: si alguna instrucción $Z(n), S(n), T(m, n, q)$ ocurre en P entonces $m, n \leq u$. Durante cualquier cómputo bajo P los registros R_v (para $v > u$) no son alterados ni tampoco afectan a los valores de $r_1, r_2, ..., r_u$. Por tanto, al construír Q los registros R_v $(v > u)$ pueden utilizarse para almacenar información utilizada por la subrutina P sin efectos colaterales. Denotaremos como $\rho(P)$ al número u.

Supóngase que P es un programa en forma estándar diseñado para computar a la función $f(x_1, ..., x_n)$. Al utilizar P como subrutina dentro de otro programa, primero las entradas $\vec{x} = x_1, ..., x_n$ deben colocarse en registros *auxiliares* $R_{l_1}, ..., R_{l_n}$ (en lugar de los primeros registros $R_1, ..., R_n$ como se ha establecido); a su vez, el resultado de $f(\vec{x})$ podría requerirse para uso posterior, por tanto deba colocarse en algún R_l (en lugar del convencional R_1); y finalmente, los registros de trabajo para P $(R_1, ..., R_{\rho(P)})$ podrían contener información no deseada.

Para ello, seguiremos la siguiente notación: $P[l_1, ...l_n \to l]$ representa el programa descrito en la Figura 2.1, el cual tiene el efecto de computar $f(r_{l_1}, ..., r_{l_n})$ y coloca el resultado en R_l. Los únicos registros afectados por éste programa son (a lo más) $R_1, R_2, ..., R_{\rho(P)}$ y R_l. Asumimos que $R_{l_1}, ..., R_{l_n}$ son distintos de $R_1, R_2, ..., R_n$.

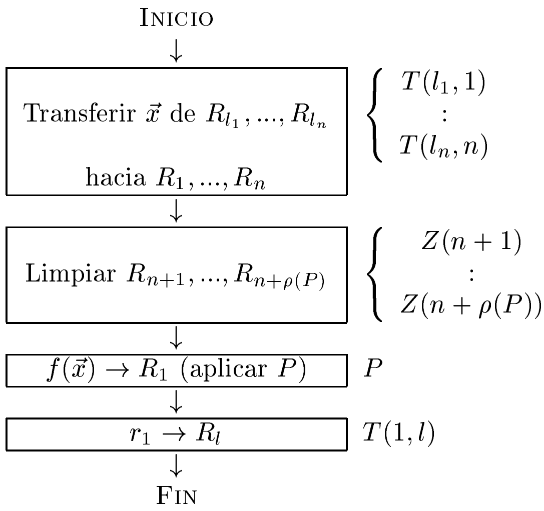

Figura 2.1: Diagrama de flujo para $P[l_1, ...l_n \to l]$.

2.3 Substitución

Una manera convencional para obtener nuevas funciones computables a partir de otras es mediante substitución ó composición de funciones. Un poco más adelante veremos que la clase de funciones computables, $\mathcal{C}$, es cerrada bajo la operación de substitución.

Teorema 1 *Supóngase que $f(y_1, ..., y_k)$ y $g_1(\vec{x}), ..., g_k(\vec{x})$ son funciones computables (donde $\vec{x} = (x_1, ..., x_n)$). La función $h(\vec{x})$ definida por*

$$h(\vec{x}) \simeq f\big(g_1(\vec{x}), ..., g_k(\vec{x})\big)$$

es computable.

Nota: $h(\vec{x})$ está definida[1] si y sólo si $g_1(\vec{x}), ..., g_k(\vec{x})$ están todas definidas y $\big(g_1(\vec{x}), ..., g_k(\vec{x})\big) \in dom(f)$; si f y $g_1, ..., g_k$ son totales entonces h también es total.

Demostración: Supóngase que $F, G_1, ..., G_k$ son programas en forma estándar los cuales computan $f, g_1, ..., g_k$ respectivamente. Para construír el programa H que compute a la función h, realizamos lo siguiente :
'Dado $\vec{x}$, invocar los programas $G_1, ..., G_k$ para computar en sucesión los valores $g_1(\vec{x}), ..., g_k(\vec{x})$. Invocar entonces al programa F para computar $f\big(g_1(\vec{x}), ..., g_k(\vec{x})\big)$.'

Para evitar perder a $\vec{x}$ y a los resultados parciales $g_i(\vec{x})$ consideremos a $m = max\big(n, k, \rho(F), \rho(G_1), ..., \rho(G_k)\big)$; primero se almacenará $\vec{x}$ en $R_{m+1}, ..., R_{m+n}$; los registros $R_{m+n+1}, ..., R_{m+n+k}$ serán utilizados para almacenar los valores

[1]Notación: utilizaremos el símbolo $\simeq$ para expresar definición.

$g_i(\vec{x})$ conforme son computados (para $i = 1, 2, ..., k$). Tales registros no son afectados por los cómputos bajo los programas $F, G_1, ..., G_k$.

Una configuración típica del cómputo bajo H es

		Registros para almacenamiento					
$R_1 \dots R_m$	$R_{m+1} \dots R_{m+n}$	R_{m+n+1}	R_{m+n+2}	...	R_{m+n+k}		
$\dots$	$\vec{x}$	$g_1(\vec{x})$	$g_2(\vec{x})$	...	$g_i(\vec{x})$	0	$\dots$

En la Figura 2.2 se muestra el diagrama del programa H que computa h:

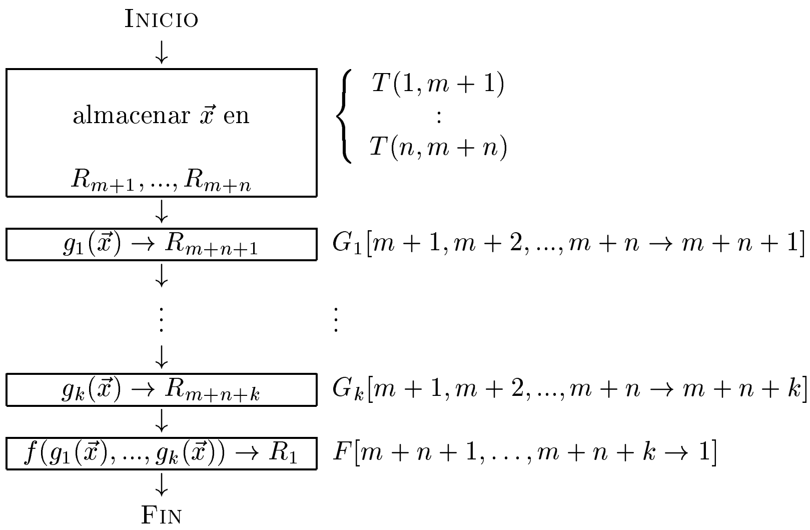

Figura 2.2: Diagrama de flujo para H.

Claramente, $H(\vec{x}) \downarrow$ *si y sólo si* cada $G_i(\vec{x}) \downarrow$ y $F(g_1(\vec{x}), ..., g_k(\vec{x})) \downarrow$.

$\diamond$

A su vez, nuevas funciones se pueden obtener a partir de cualquier función computable dada $f(z_1, z_2)$ por las siguientes manipulaciones:

$h_1(x_1, x_2) \simeq f(x_2, x_1)$	reacomodo de sus argumentos,
$h_2(x) \simeq f(x, x)$	identificación,
$h_3(x_1, x_2, x_3) \simeq f(x_2, x_3)$	variable boba.

A continuación se demuestra que cualquiera de las operaciones anteriores (o una combinación de las mismas) produce funciones computables a partir de funciones computables

Teorema 2 *Supóngase que $f(y_1, ..., y_k)$ es computable y que $x_{i_1}, ..., x_{i_k}$ es una secuencia de k de las variables $x_1, \dots, x_n$ (posiblemente con repeticiones). La función h dada por*

$$h(x_1, \dots, x_n) \simeq f(x_{i_1}, ..., x_{i_k})$$

es computable.

Demostración: podemos describir h como

$$h(\vec{x}) \simeq f\big(U^n_{i_1}(\vec{x}), \ldots, U^n_{i_k}(\vec{x})\big)$$

la cual es computable por el teorema anterior y el lema 1.
$\diamond$

Ejercicios 5 *1. Sin necesidad de escribir un programa, muestre que para todo $m \in \mathcal{N}$ las siguientes funciones son computables:*

(a) $\mathbf{m}$ *(recuerde que $\mathbf{m}(x) = m$, para toda x)*

(b) mx

2. Supónga que $f(x, y)$ es computable y $m \in \mathcal{N}$. Demuestre que la función $h(x) \simeq f(x, \mathbf{m})$ es computable.

3. Suponga que $g(x)$ es total y computable. Muestre que el predicado $M(x, y)$ dado por $M(x, y) \equiv$ '$g(x) = y$' es decidible.

2.4 Recursión

La recursión es un método para definir una función especificando cada uno de sus valores en términos de valores previamente definidos, y posiblemente utilizando otras funciones ya definidas. Para ser precisos, supóngase que $f(\vec{x})$ y $g(\vec{x}, y, z)$ son funciones (no necesariamente totales o computables). Considere a la siguiente 'definición' de una nueva función $h(\vec{x}, y)$

$$\begin{array}{ll} (i) & h(\vec{x}, 0) \simeq f(\vec{x}), \\ (ii) & h(\vec{x}, y+1) \simeq g\big(\vec{x}, y, h(\vec{x}, y)\big). \end{array} \tag{2.1}$$

La función h definida de ésta forma se dice que está definida por *recursión primitiva* a partir de las funciones f y g, las ecuaciones 2.1 se conocen como *ecuaciones de recursión.* A menos que f y g sean ambas totales h será total, el dominio de h satisface las condiciones:

$$\begin{array}{lll} (\vec{x}, 0) \in dom(h) & \textit{si y sólo si} & \vec{x} \in dom(f), \\ (\vec{x}, y+1) \in dom(h) & \textit{si y sólo si} & (\vec{x}, y) \in dom(h) \\ & & \text{y } \big(\vec{x}, y, h(\vec{x}, y)\big) \in dom(g). \end{array}$$

Resumiremos lo anterior en el siguiente teorema, cuya prueba se omite:

Teorema 3 (Recursión Primitiva) *Sea $\vec{x} = (x_1, ..., x_n)$, y sean funciones $f(\vec{x})$ y $g(\vec{x}, y, z)$, entonces existe una única función $h(\vec{x}, y)$ que satisface las siguientes ecuaciones de recursión*

$$\begin{array}{l} h(\vec{x}, 0) \simeq f(\vec{x}), \\ h(\vec{x}, y+1) \simeq g\big(\vec{x}, y, h(\vec{x}, y)\big). \end{array}$$

Cuando $n = 0$, las ecuaciones anteriores toman la forma:

$$h(0) \simeq a,$$
$$h(y+1) \simeq g\big(y, h(y)\big).$$

para $a \in \mathcal{N}$.

Ejemplo 7 1. *Suma aritmética: para toda $x, y \in \mathcal{N}$:*

$$x + 0 = x,$$
$$x + (y+1) = (x+y) + 1.$$

donde $f(x) = x$, $g(x, y, z) = z + 1$.

2. *$y!$ (el factorial):*

$$0! = 1,$$
$$(y+1)! = y!(y+1).$$

donde $a = 1$, $g(y, z) = z(y+1)$.

El siguiente resultado muestra que $\mathcal{C}$ es cerrada bajo recursión primitiva:

Teorema 4 *Dadas $f(\vec{x})$ y $g(\vec{x}, y, z)$ funciones computables, la función $h(\vec{x}, y)$ obtenida por recursión a partir de f y g, es computable.*

Demostración: Sean F y G programas en forma estándar los cuales computan a las funciones $f(\vec{x})$ y $g(\vec{x}, y, z)$. Describiremos un programa H para computar $h(\vec{x}, y)$ por recursión a partir de las ecuaciones 2.1.

Dada una configuración inicial $x_1, ..., x_n, 0, 0, ...$, H computará primero el valor $h(\vec{x}, 0)$ (aplicando F) y luego, si $y \neq 0$, H utilizará G para computar sucesivamente los valores $h(\vec{x}, 1), h(\vec{x}, 2), ..., h(\vec{x}, y)$, y entonces se detendrá.

Sea $m = max(n+1, \rho(F), \rho(G))$; se almacenan $\vec{x}, y$ en $R_{m+1}, ..., R_{m+n+1}$; los siguientes dos registros serán utilizados para almacenar el valor actual de los números k y $h(\vec{x}, k)$ para $k = 0, 1, 2, ..., y$. Sea $t = m + n$; la configuración típica del cómputo es

Registros para almacenamiento

$R_1 \dots R_m$	$R_{m+1} \dots R_t$	R_{t+1}	R_{t+2}	R_{t+3}	
$\dots$	$\vec{x}$	y	k	$h(\vec{x}, k)$	$\dots$

con $k = 0$ inicialmente.

La Figura 2.3 muestra un diagrama de flujo para éste procedimiento, el cual describe el siguiente programa H que computa a h:

$$
\begin{array}{ll}
 & T(1, m+1) \\
 & \vdots \\
 & T(n+1, m+n+1) \\
 & F[1, 2, .., n \to t+3] \\
I_q & J(t+2, t+1, p) \\
 & G[m+1, ..., m+n, t+2, t+3 \to t+3] \\
 & S(t+2) \\
 & J(1, 1, q) \\
I_p & T(t+3, 1)
\end{array}
$$

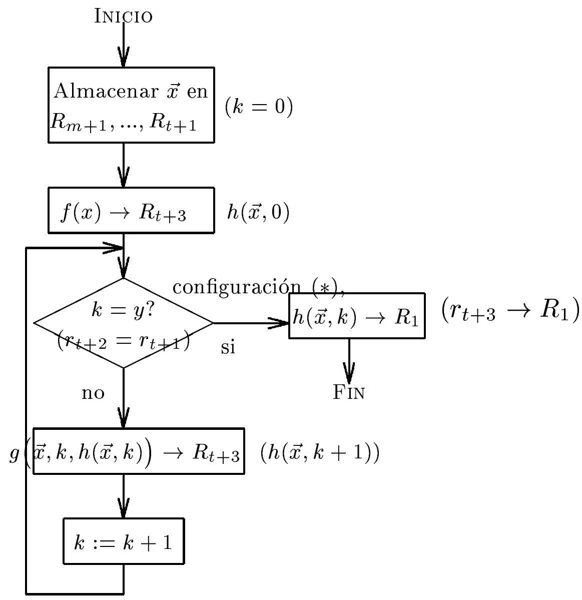

Figura 2.3: Diagrama de flujo para Recursión Primitiva.

Por tanto, h es computable.

$\diamond$

Como resultado de los teoremas anteriores mostraremos toda una colección de funciones computables, la cual es potencialmente infinita.

Teorema 5 (Biblioteca de funciones) *Las siguientes funciones son computables:*

1. $x \times y$*: por recursión utilizando a las funciones (cero)* $\mathbf{0}(x)$ *y (suma)* $z + x$ *en*

$$x \times 0 = 0$$
$$x \times (y+1) = (x \times y) + x.$$

2. x^y*: por recursión utilizando el anterior:*

$$x^0 = 0$$
$$x^{(y+1)} = x^y \times x.$$

3. $x \overset{\circ}{-} 1$*: (diferencia positiva) por recursión*

$$0 \overset{\circ}{-} 1 = 0$$
$$(x+1) \overset{\circ}{-} 1 = x.$$

4. $x \overset{\circ}{-} y = \begin{cases} x - y & \textit{si } x \geq y, \\ 0 & \textit{en otro caso.} \end{cases}$ *por recursión aplicando el anterior:*

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

5. $sg(x) = \begin{cases} 0 & \textit{si } x = 0, \\ 1 & \textit{si } x \neq 0 \end{cases}$ *por recursión:*

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

6. $\overline{sg}(x) = \begin{cases} 1 & \textit{si } x = 0, \\ 0 & \textit{si } x \neq 0 \end{cases}$ *(ejercicio).*

7. $|x - y|$ *(ejercicio).*

8. $x!$ *(ejercicio).*

9. $min(x, y)$*: por substitución*

$$min(x, y) = x \overset{\circ}{-} (x \overset{\circ}{-} y)$$

10. $max(x, y)$ *(ejercicio).*

11. $rm(x, y)$ *el residuo de* y *sobre* x *(convención* $rm(0, y) = y$*):*

$$rm(x, y+1) = \begin{cases} rm(x, y) + 1 & \textit{si } rm(x, y) + 1 \neq x, \\ 0 & \textit{si } rm(x, y) + 1 = x \end{cases}$$

la cual produce la siguiente definición por recursión:

$$\begin{aligned} & rm(x, 0) = 0 \\ & rm(x, y+1) = \big(rm(x, y) + 1\big) \times sg\big(|x - \big(rm(x, y) + 1\big)|\big). \end{aligned}$$

12. $qt(x, y)$ *el cociente de* y *sobre* x *(convención* $qt(0, y) = 0$*):*

$$qt(x, y+1) = \begin{cases} qt(x, y) + 1 & \textit{si } rm(x, y) + 1 = x, \\ qt(x, y) & \textit{si } rm(x, y) + 1 \neq x \end{cases}$$

ahora tenemos la siguiente definición por recursión:

$$\begin{aligned} & qt(x, 0) = 0 \\ & qt(x, y+1) = qt(x, y) + \overline{sg}\big(|x - \big(rm(x, y) + 1\big)|\big). \end{aligned}$$

13. $div(x, y) = \begin{cases} 1 & \textit{si } x/y (x \textit{ divide a } y), \\ 0 & \textit{de lo contrario.} \end{cases}$

(convención: $0/y$ *únicamente si* $y = 0$*). Por substitución:*

$$div(x, y) = \overline{sg}\big(rm(x, y)\big).$$

$\diamond$

Los siguientes corolarios involucran predicados decidibles:

Corolario 1 (Definición por casos) *Supóngase que $f_1(\vec{x}), \ldots, f_k(\vec{x})$ son funciones computables y totales. Sean $M_1(\vec{x}), \ldots, M_k(\vec{x})$ predicados decidibles tales que para toda $\vec{x}$ exactamente uno de ellos se cumple. La función $g(\vec{x})$ dada por*

$$g(\vec{x}) = \begin{cases} f_1(\vec{x}) & si\ M_1(\vec{x})\ se\ cumple, \\ f_2(\vec{x}) & si\ M_2(\vec{x})\ se\ cumple, \\ \vdots & \vdots \\ f_k(\vec{x}) & si\ M_k(\vec{x})\ se\ cumple. \end{cases}$$

es computable.

Demostración: construyamos a g aplicando substitución, suma y producto:

$$g(\vec{x}) = c_{M_1}(\vec{x}) \times f_1(\vec{x}) + \ldots + c_{M_k}(\vec{x}) \times f_k(\vec{x})$$

$\diamond$

Corolario 2 (Álgebra de Decidibilidad) *Sean $M(\vec{x})$ y $Q(\vec{x})$ predicados decidibles. Los siguientes también son predicados decidibles:*

1. *'no $M(\vec{x})$'*
2. *'$M(\vec{x})$ y $Q(\vec{x})$'*
3. *'$M(\vec{x})$ o $Q(\vec{x})$'*

Demostración: las funciones características de tales predicados se pueden construír de la siguiente manera:

1. 'no $M(\vec{x})$': $1 \overset{\circ}{-} c_M(\vec{x})$
2. '$M(\vec{x})$ y $Q(\vec{x})$': $c_M(\vec{x}) \times c_Q(\vec{x})$
3. '$M(\vec{x})$ o $Q(\vec{x})$': $max\big(c_M(\vec{x}), c_Q(\vec{x})\big)$ (se considera 'o' incluyente).

$\diamond$

Las funciones computables totales que se pueden construír a partir de las funciones primitivas utilizando substitución y recursión se denominan **funciones primitivas recursivas**.

2.4.1 Sumas y Productos acotados

A continuación se introducen otras técnicas para construcción de funciones.

Definición 7 *Sea $f(\vec{x}, z)$ cualquier función, la* **suma acotada** $\sum_{z<y} f(\vec{x}, z)$ *y el* **producto acotado** $\prod_{z<y} f(\vec{x}, z)$ *son las funciones de los argumentos $\vec{x}, y$ definidas por las siguientes ecuaciones recursivas:*

$$\begin{cases} \sum_{z<0} f(\vec{x}, z) = 0, \\ \sum_{z<y+1} f(\vec{x}, z) = \sum_{z<y} f(\vec{x}, z) + f(\vec{x}, z) \end{cases} \tag{2.2}$$

$$\begin{cases} \prod_{z<0} f(\vec{x}, z) = 1, \\ \prod_{z<y+1} f(\vec{x}, z) = \left(\prod_{z<y} f(\vec{x}, z)\right) \times f(\vec{x}, z) \end{cases} \tag{2.3}$$

Teorema 6 *Si $f(\vec{x}, z)$ es total y computable, entonces las funciones $\sum_{z<y} f(\vec{x}, z)$ y $\prod_{z<y} f(\vec{x}, z)$ también son totales y computables.*

Demostración: ejercicio.

A continuación, se podrá apreciar que si la cota en z en una suma ó producto acotado se da para cualquier función computable, el resultado también es computable.

Corolario 3 *Supóngase que $f(\vec{x}, z)$ y $k(\vec{x}, \vec{w})$ son totales y computables. Las funciones $\sum_{z<k(\vec{x},\vec{w})} f(\vec{x}, z)$ y $\prod_{z<k(\vec{x},\vec{w})} f(\vec{x}, z)$ también son totales y computables (ambas toman como argumentos a $\vec{x}, \vec{w}$).*

Demostración: aplicando substitución de $k(\vec{x}, \vec{w})$ para y en la suma acotada $\sum_{z<y} f(\vec{x}, z)$ y en el producto $\prod_{z<y} f(\vec{x}, z)$.
$\diamond$

Otra técnica muy útil para construír funciones computables es el **Operador de minimización acotada** μ que definimos a continuación: escribimos

$$\mu z < y(\ldots)$$

que se lee '*el z más pequeño menor que y tal que ...*'.

Por ejemplo, dada una función $f(\vec{x}, z)$ se puede definir una función g por

$$\begin{aligned} g(\vec{x}, y) &= \mu z < y\big(f(\vec{x}, z)\big) \\ &= \begin{cases} \text{el menor } z < y & \text{tal que } f(\vec{x}, z) = 0, \text{si tal } z \text{ existe;} \\ y & \text{si no existe tal } z. \end{cases} \end{aligned}$$

Teorema 7 *Supóngase que $f(\vec{x}, z)$ es total y computable, entonces la función $\mu z < y\big(f(\vec{x}, z) = 0\big)$ también es total y computable.*

Demostración: considérese a la función

$$h(\vec{x}, v) = \prod_{u \leq v} sg\big(f(\vec{x}, u)\big)$$

la cual es computable por el corolario 3; para los argumentos dados $\vec{x}, y$ supóngase que $z_0 = \mu z < y\big(f(\vec{x}, z) = 0\big)$ se puede apreciar que

$$\begin{gathered} \text{si } v < z_0 \text{ entonces } h(\vec{x}, v) = 1; \\ \text{si } z_0 \leq v < y \text{ entonces } h(\vec{x}, v) = 0. \end{gathered}$$

Por tanto, z_0 es el número de v's menores que y tales que hacen $h(\vec{x}, v) = 1$, i.e., $z_0 = \sum_{v<y} f(\vec{x}, v)$. De aquí, podemos expresar $\mu z < y\big(f(\vec{x}, z) = 0\big)$ mediante sumatoria y producto acotado de funciones computables:

$$\mu z < y\big(f(\vec{x}, z) = 0\big) = \sum_{v<y} \left(\prod_{u \leq v} sg\big(f(\vec{x}, u)\big) \right)$$

$\diamond$

La cota y en una expresión de minimización acotada $\mu z < y(...)$ también puede ser una función computable:

Corolario 4 *Si $f(\vec{x}, z)$ y $k(\vec{x}, \vec{w})$ son funciones totales y computables, también lo es la función*

$$\mu z < k(\vec{x}, \vec{w})\big(f(\vec{x}, z) = 0\big).$$

Demostración: simplemente aplicamos substitución de $k(\vec{x}, \vec{w})$ por y en $\mu z < y\big(f(\vec{x}, z) = 0\big)$.

$\diamond$

Como consecuencia, tenemos los siguientes resultados

Corolario 5 *Sea $R(\vec{x}, y)$ un predicado decidible, entonces:*

1. *la función $f(\vec{x}, y) = \mu z < yR(\vec{x}, z)$ es computable.*
2. *los siguientes predicados son decidibles:*
   (a) $M_1(\vec{x}, y) \equiv \forall z < yR(\vec{x}, z)$,
   (b) $M_2(\vec{x}, y) \equiv \exists z < yR(\vec{x}, z)$.

Demostración:

1. Expresamos a f como $f(\vec{x}, y) = \mu z < y\big(\overline{sg}(C_R(\vec{x}, z)) = 0\big)$ claramente es computable.
2. (a) Definimos la función característica de M_1 por $c_{M_1}(\vec{x}, y) = \prod_{z<y} c_R(\vec{x}, z)$.
   (b) Ahora aprovechamos el resultado anterior: $M_2(\vec{x}, y) \equiv no\big(\forall z < y\big(noR(\vec{x}, z)\big)\big)$.

$\diamond$

Utilizando las técnicas expuestas, podemos agrandar la biblioteca de funciones computables y predicados decidibles:

Teorema 8 *Las siguientes funciones son computables:*

1. *$D(x) =$ el número de divisores de x (convención: $D(0) = 1$).*
2. $Pri(x) = \begin{cases} 1 & \textit{si } x \textit{ es primo}, \\ 0 & \textit{si } x \textit{ no es primo}. \end{cases}$
3. *$p_x =$ el x-ésimo número primo (convemción: $p_0 = 0$).*
4. $(x)_y = \begin{cases} \textit{el exponente de } p_y & \textit{si } x, y > 0, \\ \textit{en la factorización prima de } x & \\ 0 & \textit{si } x = 0 \textit{ ó } y = 0. \end{cases}$

Demostración: ejercicio.

La función $(x)_y$ se utiliza como una forma de *codificación* al estilo de la criptografía: una secuencia $\vec{s} = (a_1, a_2, \ldots, a_n) \in \mathcal{N}^n$ se puede codificar mediante un único número $b = p_1^{a_1+1} \times p_2^{a_2+1} \times \ldots \times p_n^{a_n+1}$; entonces, la longitud n de s y los números a_i se pueden recuperar de manera efectiva a partir de b de la siguiente manera:

$$\begin{aligned} n = &\quad \mu z < b\big((b)_{z+1} = 0\big) \\ a_i = &\quad (b)_i \mathbin{\dot{-}} 1, \text{ para } 1 \leq i \leq n. \end{aligned}$$

La codificación es esencial para los temas que discutiremos más adelante.

Ejercicios 6 *1. Demuestre que las siguientes funciones son computables:*

(a) Cualquier función polinómica $a_0 + a_1x + \ldots + a_nx^n$ *para* $\vec{a} \in \mathcal{N}$.

(b) $[\sqrt{x}]$

(c) $LCM(x, y)$ *= mínimo común múltiplo de* x, y.

(d) $HCF(x, y)$ *= máximo común divisor de* x, y.

(e) $f(x)$ *= número de divisores primos de* x.

(f) Función de Euler: $\phi(x)$ *= número de enteros positivos menores que* x *los cuales son primos relativos a* x. *Decimos que* x, y *son primos relativos si* $HCF(x, y) = 1$.

2. Sea $\pi(x, y) = 2^x \times (2 \times y + 1) - 1$. *Demuestre que* π *es una biyección computable de* $\mathcal{N}^2 \longrightarrow \mathcal{N}$, *y que las funciones* π_1, π_2 *tales que la composición* $\pi(\pi_1(z), \pi_2(z)) = z$ *(para cualquier* z*) también son computables.*

3. Secuencia Fibonacci: demostrar que la función $f(x)$ *definida por*

$$f(x) = \begin{cases} f(0) = 1, \\ f(1) = 1, \\ f(x+2) = f(x) + f(x+1). \end{cases}$$

es computable.

4. Demostrar que los siguientes problemas son decidibles:

(a) x *es impar,*

(b) x *es una potencia de un número primo,*

(c) x *es un cubo perfecto.*

5. (Otra forma de ciframiento:) cualquier número $\alpha \in \mathcal{N}$ *tiene una expresión única definida por*

(a) $x = \sum_{i=0}^{\infty} \alpha_i \times 2^i$, *donde para toda* i, α_i *sólo puede tomar valores* 0 *ó* 1.

Si $x > 0$ *existen expresiones únicas para* x *de la forma:*

(b) $x = 2^{b_1} + 2^{b_2} + \ldots + 2^{b_l}$, *donde* $0 \leq b_1 < b_2 < \ldots < b_l$ *para* $l \geq 1$. *A su vez, otra expresión para* x *es*

(c) $x = 2^{a_1} + 2^{a_1+a_2+1} + \ldots + 2^{a_1+a_2+\ldots+a_k+k+1}$

Se establecen a continuación las siguientes funciones

(a) $\alpha(i, x) = \alpha_i$, *como en la primer expresión para* x *de arriba.*

(b) $\mathbf{l}(x) = \begin{cases} l & \text{(como en la segunda expresión arriba) si } x > 0, \\ 0 & \text{en otro caso.} \end{cases}$

(c) $\mathbf{b}(i, x) = \begin{cases} b_i & \text{(como en la segunda expresión arriba) si } x > 0, 1 \leq i \leq l \\ 0 & \text{en otro caso.} \end{cases}$

(d) $\mathbf{a}(i, x) = \begin{cases} a_i & \text{(como en la tercera expresión arriba) si } x > 0, 1 \leq i \leq l \\ 0 & \text{en otro caso.} \end{cases}$

(Nótese que se está jugando con los números para describirlos en términos de funciones computables).

Demuestre que cada una de las funciones $\alpha, \mathbf{l}, \mathbf{b}, \mathbf{a}$ *es computable.*

2.5 Minimización

En la sección anterior pudimos demostrar que una gran colección de funciones son computables utilizando las operaciones de substitución, recursión y las operaciones derivadas de ellas. Discutiremos a continuación una tercera operación importante que permite generar más funciones computables (a partir de funciones computables), denominada *minimización acotada* que es una generalización natural del operador μ anterior.

Conviene primero plantear la idea: sea $f(\vec{x}, y)$ una función (no necesariamente total). Deseamos definir una función $g(\vec{x})$ como

$$g(\vec{x}) = \text{ el menor } y \text{ tal que } f(\vec{x}, y) = 0.$$

de tal forma que si f es computable entonces g también lo es.

Dos problemas tienen lugar:

1. para algún $\vec{x}$ podría no existir algún y tal que $f(\vec{x}, y) = 0$.

2. Asumiendo que f sea computable, considérese el siguiente algoritmo para computar $g(\vec{x})$:
   'Calcular $f(\vec{x}, 0), f(\vec{x}, 1), f(\vec{x}, 2), \ldots$ hasta que se encuentre algún y tal que $f(\vec{x}, y) = 0$.'

   Éste procedimiento podría no terminar si f no es total, aún si existiese esa y; por ejemplo, si $f(\vec{x}, 0)$ no está definida (i.e., el programa no termina al evaluar en $(\vec{x}, 0)$) pero $f(\vec{x}, 1) = 0$. El programa no tiene forma de averiguarlo, tiene que revisar en orden los posibles argumentos para f.

Esta idea nos conduce a la siguiente definición para el operador de minimización μ:

Definición 8 *Para cualquier función $f(\vec{x}, y)$*

$$\mu y\big(f(\vec{x}, y) = 0\big) = \begin{cases} \textit{el menor} & y \textit{ tal que} \\ & (i) f(\vec{x}, z) \textit{ está definida} \quad \textit{para toda } z \leq y, \\ & (ii) f(\vec{x}, z) = 0, \quad \textit{si tal } y \textit{ existe.} \\ \textit{indefinida} & \textit{si no existe tal } y. \end{cases}$$

$\mu y(\ldots)x$ se lee 'el menor y tal que $\ldots$', el cual simplemente se le denomina operador μ.

Mostraremos a continuación que $\mathcal{C}$ es cerrada bajo minimización

Teorema 9 *Sea $f(\vec{x}, y)$ computable, entonces la función*

$$g(\vec{x}) = \mu y\big(f(\vec{x}, y) = 0\big)$$

también es computable.

Demostración:

Sea $\vec{x} = (x_1, x_2, \ldots, x_n)$ y sea F un programa en forma estándar que computa a la función $f(\vec{x}, y)$. Sea $m = max(n+1, \rho(F))$. Construiremos un programa G que compute g: para $k = 0, 1, 2, \ldots$ calcúlese $f(\vec{x}, k)$ hasta que algún valor de k se encuentre tal que $f(\vec{x}, y) = 0$; éste valor de k es el resultado requerido.

El valor de $\vec{x}$ y el valor actual de k serán almacenados en los registros $R_{m+1}, \ldots, R_{m+n+1}$ antes de computar $f(\vec{x}, k)$, por tanto la configuración típica será

Registros para almacenamiento

$R_1 \ldots R_m$	$R_{m+1} \ldots R_{m+n}$	R_{m+n+1}	R_{m+n+2}	$\ldots$
$\ldots$	$\vec{x}$	k	0	$\ldots$

con $k = 0$ inicialmente. Nótese que el valor r_{m+n+2} siempre es 0. Se muestra el programa G a continuación, así como su diagrama de flujo en la Figura 2.4.

$$\begin{array}{ll} & T(1, m+1) \\ & \vdots \\ & T(n, m+n) \\ I_p & F[m+1, m+2, .., m+n+1 \rightarrow 1] \\ & J(1, m+n+2, q) \\ & S(m+n+1) \\ & J(1,1,p) \\ I_q & T(m+n+1, 1) \end{array}$$

Por tanto, g es computable.

$\diamond$

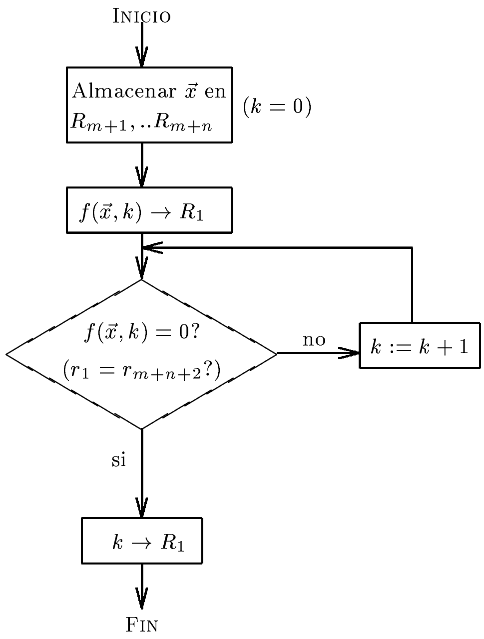

Figura 2.4: Diagrama de flujo para el Teorema de Minimización.

Corolario 6 *Sea $R(\vec{x},y)$ un predicado decidible. La función*

$$\begin{aligned} g(\vec{x}) &= \mu y R(\vec{x},y) \\ &= \begin{cases} \textit{el menor } y \textit{ tal que } R(\vec{x},y) \textit{ se cumple,} & \textit{si existe tal } y, \\ \textit{indefinido} & \textit{de lo contrario.} \end{cases} \end{aligned}$$

es computable.

Demostración: $g(\vec{x}) = \mu y\big(\overline{sg}\big(c_R(\vec{x},y)\big) = 0\big)$.

$\diamond$

Al operador μ a menudo se le denomina *operador de búsqueda*: dado un predicado decidible $R(\vec{x},y)$ la función $g(\vec{x})$ busca por un y tal que $R(\vec{x},y)$ sea verdadero, y más aún, encuentra el menor y si existe uno.

El operador μ puede generar una función computable no total a partir de una función computable total; por ejemplo, sean $f(x,y) = |x - y^2|$ y defínase a $g(x) \simeq \mu y\big(f(x,y) = 0\big)$, tenemos que g no es total:

$$g(x) = \begin{cases} \sqrt{x} & \text{si } x \text{ es cuadrado perfecto,} \\ \text{indefinido} & \text{en otro caso.} \end{cases}$$

Minimización, a diferencia de minimización acotada, no puede definirse en términos de substitución y recursión. Existen funciones totales para las cuales es indispensable el operador de minimización μ (por ejemplo, la función de Ackermann que veremos en un momento), por tanto ya no pertenecen a la clase de funciones primitivas recursivas.

Ejercicios 7 *1. Sea $f(x)$ inyectiva, total y computable, demostrar que f^{-1} es computable.*

2. *Sea $p(x)$ un polinomio con coeficientes enteros, mostrar que la función $f(a)$ = la menor raíz integral no negativa de $p(x) - a$ ($a \in \mathcal{N}$) es computable. ($f(a)$ queda indefinida si no existe tal raíz).*

3. *Mostrar que la función*

$$f(\vec{x}, y) = \begin{cases} x/y & si\ y \neq 0, div(x,y) = 1 \\ \ indefinida & de\ lo\ contrario. \end{cases}$$

Para finalizar éste capítulo se presenta a una función en la que el operador μ es esencial, también muestra que μ puede utilizarse no sólo para buscar un número que cumpla con alguna propiedad sino también para buscar por secuencias finitas (conjuntos) de números (u otros objetos) codificados por un único número.

Ejemplo 8 (Función de Ackermann) *La función $\mathcal{A}(x,y)$ dada por las siguientes ecuaciones*

$$\begin{array}{l} \mathcal{A}(0,y) = y+1, \\ \mathcal{A}(x+1,0) \simeq \mathcal{A}(x,1), \\ \mathcal{A}(x+1,y+1) \simeq \mathcal{A}\big(x, \mathcal{A}(x+1,y)\big). \end{array}$$

es computable.

Podemos apreciar que la definición de $\mathcal{A}$ involucra doble recursión la cual es más fuerte que la recursión primitiva. Nó tese que para cualquier argumento, $\mathcal{A}(x,y)$ se define en términos de valores anteriores $\mathcal{A}(x_1,y_1)$ con $x_1 < x$ ó $x_1 = x$ pero $y_1 < y$; de hecho el valor resultante de $\mathcal{A}(x,y)$ puede obtenerse utilizando únicamente un número finito de valores anteriores, lo cual se puede demostrar mediante inducción en x y y.

La demostración formal que $\mathcal{A}$ es computable es bastante difícil, por lo que se presentará una prueba informal en la que el operador μ es esencial. Para ello se hace uso de la idea de un conjunto de triples *adecuado S*. La propiedad esencial de un conjuntos adecuado S es que si $(x,y,z) \in S$, entonces

1. $z = \mathcal{A}(x,y)$,

2. S contiene a todos los triples anteriores $\big(x_1, y_1, \mathcal{A}(x_1,y_1)\big)$ que se necesitan para calcular $\mathcal{A}(x,y)$.

Definición 9 *Un conjunto de triples S se dice que es adecuado si se satisfacen las siguientes condiciones:*

1. *si* $(0, y, z) \in S$ *entonces* $z = y + 1$,

2. *si* $(x + 1, 0, z) \in S$ *entonces* $(x, 1, z) \in S$

3. *si* $(x + 1, y + 1, z) \in S$ *entonces existe* u *tal que*

$$(x + 1, y, u) \in S \quad y \quad (x, u, z) \in S.$$

Las condiciones corresponden claramente a las tres cláusulas en la definición de $\mathcal{A}$. Para cualquier pareja (m, n) existe un conjunto adecuado S tal que $\big(m, n, \mathcal{A}(m, n)\big) \in S$, basta con que S sea el conjunto de todos los triples $\big(x, y, \mathcal{A}(x, y)\big)$ que se utilizan para calcular $\mathcal{A}(m, n)$.

Un triple (x, y, z) puede codificarse por un único natural $u = 2^x \times 3^y \times 5^z$; a su vez, un conjunto finito de naturales $\{u_1, \ldots, u_k\}$ se pueden codificar mediante un único natural $p_{u_1} \times p_{u_2} \times \ldots \times p_{u_k}$. En particular, un conjunto finito de triples se puede codificar por un único natural v. Denotaremos por S_v el conjunto de triples codificados por v. Tenemos lo siguiente

$$(x, y, z) \in S_v \textit{ si y sólo si } p_u \text{ divide a } v$$

por tanto, el predicado '$(x, y, z) \in S_v$' es decidible; si se cumple para algún x, y, z, v entonces eso significa que $x, y, z < v$. Utilizando técnicas y funciones ya discutidas podemos mostrar que el siguiente predicado es decidible:

$$R(x, y, v) \equiv \text{ '}v\text{ es el número de código de un conjunto adecuado de triples y } \exists z < v\big((x, y, z) \in S_v\big)\text{'}$$

Por tanto, la función

$$f(x, y) = \mu v R(x, y, v)$$

es computable y busca el código de un conjunto adecuado que contenga al triple (x, y, z) para algún z. De aquí podemos expresar a $\mathcal{A}$ como

$$\mathcal{A}(x, y) = \mu z\big((x, y, z) \in S_{f(x,y)}\big)$$

lo cual muestra que $\mathcal{A}$ es computable.

Capítulo 3

La Tesis de Church

Dentro de los últimos 70 años se han propuesto numerosas ideas para caracterizar matemáticamente la noción intuitiva de computabilidad efectiva. El enfoque URM es uno de los más recientes. Haremos en éste capítulo una breve resemblanza de algunas de las propuestas más importantes y considerar dos preguntas muy relacionadas:

1. ¿Que tan distintos son los otros enfoques a la computabilidad (y cómo compararlos)?

2. ¿Que tan adecuados son los enfoques (en particular el modelo URM) para caracterizar la idea informal de computabilidad?

Discutiremos las ideas esenciales exclusivamente para tener una idea clara de las diferencias entre los distintos enfoques.

3.1 Otros Enfoques

A continuación se mencionan algunas de las caracterizaciones a la noción de computabilidad:

- *Gödel-Herbrand-Kleene* (1936) funciones recursivas generales definidas por medio de un cálculo ecuacional.

- Máquinas de Turing (Alan Turing, 1936) funciones computables por máquinas finitas.

- Funciones definidas en el $\lambda-$*calculus* (Alonzo Church, 1933).

- Sistemas de Post (Emil Post, 1950) funciones definidas mediante sistemas de deducción canónica.

- Funciones $\mu-$recursivas y funciones recursivas parciales (Kurt Gödel, Jacques Herbrand 1936).

- Lógica Combinatoria (Moses Schönfinkel, Haskell B. Curry 1924).
- Funciones dadas por ciertos algoritmos sobr eun alfabeto finito (Markov 1951).
- Funciones computables URM (Sheperson-Sturgis 1963).

3.1.1 El Resultado Fundamental

Cada una de las propuestas para caracterizar la noción de computabilidad efectiva describe a la misma clase de funciones, que se ha denotado como $\mathcal{C}$.

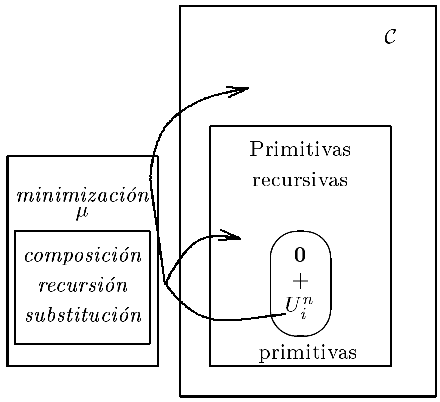

Figura 3.1: La clase de funciones computables $\mathcal{C}$ como un *conjunto inductivo.*

3.2 Computabilidad Turing

3.3 Sistemas de Post

3.4 Cálculo λ

3.5 La Tesis de Church

Enfocaremos ahora nuestra atención hacia la segunda pregunta del inicio del capítulo. La tesis de Church (Church-Turing) se puede establecer así:

La clase de funciones parciales efectivamente computables definida informalmente coincide exactamente con la clase $\mathcal{C}$ de funciones URM *computables.*

Nótese que éste resultado no es un teorema suceptible a una prueba matemática, tiene el estado de una *creencia* que debe ser sostenida por la evidencia, la cual es considerable:

1. El resultado fundamental: muchas propuestas independientes han conducido a la misma clase de funciones $\mathcal{C}$.

2. Se ha demostrado explícitamente que una vasta colección de funciones efectivamente computables pertencen a $\mathcal{C}$ (capítulo 2).

3. La implantación de un programa URM P para computar una función es un ejemplo claro de algoritmo; por lo que directamente a partir de la definición de $\mathcal{C}$ todas las funciones en $\mathcal{C}$ son computables en el sentido informal.

4. Nadie ha encontrado una función que pueda aceptarse como computable en el sentido informal y que no pertenezca a $\mathcal{C}$.

En base a la evidencia y en su propia experiencia la mayoría de los matemáticos aceptan la tesis de Church. Nosotros aceptaremos la tesis de Church en el sentido siguiente: supóngase que hemos descrito informalmente un algoritmo para computar los valores de una función f. Tal algoritmo puede describirse en lenguaje natural ó mediante diagramas ó medios semi formales o de algún otro medio que de manera no ambigua indique cómo calcular efectivamente los valores de f (donde esté definida) en tiempo finito. Para tal situación, queremos mostrar que f es URM computable, para lo cual tenemos dos métodos:

1. Escribir un programa URM que compute a f (demostrando que lo hace) ó demostrando indirectamente que tal programa existe (mediante las técnicas del capítulo anterior).

   La desventaja de éste método es que la prueba formal que f es URM computable podría ser bastante larga y tediosa. Esencialmente, involucraría una forma de *traducción* del algoritmo informal hacia el programa ó el lenguaje de alguna de las distintas caracterizaciones formales (los diagramas de flujo serían traducciones intermedias).

2. Ofrecer una prueba rigurosa (informal) de que el algoritmo sirve para computar f. Entonces apelar a la tesis de Church y concluír inmediatamente que f es URM computable (equivalentemente, computable en alguno de los otros enfoques).

Ilustraremos el segundo método (que denominaremos *prueba por la tesis de Church*) con los siguientes ejemplos

Ejemplo 9 *1. Sea P un programa* URM, *defínase la función f por*

$$f(x,y,t) = \begin{cases} 1 & \textit{si } P(x)\downarrow y \textit{ tras ejecutar } t \textit{ o menos pasos,} \\ 0 & \textit{de lo contrario.} \end{cases}$$

Podemos plantear un algoritmo informal para computar f de la siguiente forma:

'Dado (x, y, t) simular el cómputo de $P(x)$ llevando al cabo t pasos a menos que $P(x)$ se detenga en menos de t pasos y en R_1 se encuentre el valor y, en cuyo caso regresar el valor de $f(x, y, t)$ a 1.

Si $P(x)$ no se ha detenido tras t pasos (o ya se detuvo pero el valor en R_1 es distinto de y) entonces regresar el valor de $f(x, y, t)$ a 0.'

Por la tesis de Church, f es URM computable dado que la simulación de $P(x)$ para a lo más t pasos puede completarse en una cantidad finita de tiempo.

2. *Supóngase que f y g son computables. Defínase h por*

$$h(x) = \begin{cases} 1 & si\ x \in Dom(f)\ ó\ x \in Dom(g), \\ indefinida & en\ cualquier\ otro\ caso. \end{cases}$$

Podemos describir un algoritmo para computar h de la siguiente forma:

'Dado x, iniciar los programas correspondientes a $f(x)$ y $g(x)$ simultáneamente. Si alguno de los programas termina, detenerse y establecer que $h(x) = 1$; en otro caso, continuar de manera indefinida.'

Éste algoritmo regresa $h(x) = 1$ para cualquier x tal que ya sea $f(x)$ ó $g(x)$ estén definidas, y continúa indefinidamente si ninguna está definida en x. Por la tesis de Church, h es computable.

Sin embargo, el lector debe proporcionar pruebas completas (por el método 1) que las funciones son URM computables.

Al utilizar la tesis de Church no se propone abandonar la construcción de pruebas como si la tesis de Church fuese una varita mágica. La tesis evita que la idea fundamental de la demostración de que si una función en computable se pierda entre detalles técnicos intricados. No obstante, queda como una expresión de fé o confianza.

En conclusión: *en adelante, aceptaremos la tesis de Church y la utilizaremos en la forma de prueba descrita* (omitiendo a menudo referencia explícita).

Ejercicios 8 1. *Sean $f(x)$ y $g(x)$ computables. Demostrar por la tesis de Church que la función h dada por*

$$h(x) = \begin{cases} x & si\ x \in Dom(f) \cap Dom(g), \\ indefinida & en\ cualquier\ otro\ caso. \end{cases}$$

es URM computable.

2. *Sea f unaria, total y computable. Demostrar que la función h*

$$h(x) = \begin{cases} 1 & si\ x \in Rango(f), \\ indefinida & en\ cualquier\ otro\ caso. \end{cases}$$

es URM computable.

3. *Aplicar la tesis de Church para demostrar que la función de Ackermann es computable.*

4. *Demostrar que la función g*

$$g(n) = n\text{-\textit{ésimo dígito en la expansión decimal de} } e$$

es computable (donde e es la base de los logaritmos naturales).

Capítulo 4

Enumerando Funciones Computables

Continuamos con el estudio de las funciones URM computables; por tanto, la expresión *computable* se referirá URM computable y un *programa* significará un programa URM.

El aspecto fundamental de éste capítulo es que el conjunto de todos los programas puede *enumerarse efectivamente*: existe una codificación efectiva de los programas por el conjunto de los naturales, lo cual significa que $\mathcal{C}$ también es enumerable y con ello que existan muchas funciones que no son computables, lo cual puede demostrarse por el método de la diagonalización de Cantor

El enumerado ó codificación de los programas, y en particular el hecho que tal enumeramiento sea efectivo (i.e, se puede plantear un algoritmo para enumerar los programas) es un aspecto fundamental para el desarrollo de la teoría de computabilidad. Como resultado podemos representar a las funciones mediante un número (su código ó **índice**) y por tanto podemos considerar la idea de realizar operaciones efectivas involucrando tales códigos.

4.1 Enumerando Programas

Comenzaremos introduciendo la terminología

Definición 10 (Conjunto Denumerable) *Un conjunto X se dice* **denumerable** *(contable) si existe una biyección $f : X \longrightarrow \mathcal{N}$.*

Definición 11 (Enumeración) *La enumeración de conjunto X es una función sobreyectiva $g : \mathcal{N} \longrightarrow X$, lo cual se representa simplemente subindicando los elementos de X: $\{x_0, x_1, \ldots\}$*
donde $x_n = g(n)$. Se dice que g es una enumeración sin repeticiones si g es inyectiva.

Definición 12 (Enumeración efectiva) *Sea X un conjunto finito de objetos (números, programas, instrucciones, etc.) se dice que X es* **efectivamente denumerable**[1] *si existe una biyección $f : X \longrightarrow \mathcal{N}$ tal que tanto f como f^{-1} son computables.*

Teorema 10 *Los siguientes conjuntos son efectivamente denumerables*[2]*:*

1. *$\mathcal{N} \times \mathcal{N}$ (parejas ordenadas del producto cartesiano)*
2. *$\mathcal{N}^+ \times \mathcal{N}^+ \times \mathcal{N}^+$ (triples)*
3. *$\bigcup_{k>0} \mathcal{N}^k$ (el conjunto de todas las secuencias finitas de naturales).*

Demostración:

1. Se define la biyección $\pi : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ por

   $$\pi(m, n) = 2^m(2n + 1) - 1$$

   se puede apreciar inmediatamente que π es computable, para ver que su inversa también lo es, obsérvese que:

   $$\pi^{-1}(x) = \big(\pi_1(x), \pi_2(x)\big)$$

   donde π_1, π_2 son computables definidas por

   $$\begin{aligned} \pi_1(x) = &\ \lg_2\big(\tfrac{z+1}{\pi_2}\big) \\ &\ \text{donde } z = 2^x(2y + 1) - 1 \\ \pi_2(x) = &\ \tfrac{1}{2}\big(\tfrac{(x+1)}{2^{\pi_1(x)} - 1}\big) \end{aligned}$$

2. La biyección $\varsigma : \mathcal{N}^+ \times \mathcal{N}^+ \times \mathcal{N}^+ \to \mathcal{N}$ (y su inversa) se construye a partir de π:

   $$\begin{aligned} &\varsigma(m, n, q) = \pi\big(\pi(m - 1, n - 1), q - 1\big). \\ \text{por tanto:}\quad &\varsigma^{-1}(x) = \big(\pi_1\big(\pi_1(x)\big) + 1, \pi_2\big(\pi_2(x)\big) + 1, \pi_2(x) + 1\big). \end{aligned}$$

3. Se define la biyección $\tau : \bigcup_{k>0} \mathcal{N}^k \to \mathcal{N}$ como

   $$\begin{aligned} \tau(a_1, \dots, a_k) = &\ 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + \dots \\ &+ 2^{a_1+a_2+\dots+a_k+k+1} - 1. \end{aligned}$$

   τ es computable. Verifiquemos que τ es una biyección y que τ^{-1} es computable: recordemos que para cada natural x podemos encontrar efectivamente números únicos $k \geq 1$ y $0 \leq b_1 < b_2 < \dots < b_k$ tales que

   $$x + 1 = 2^{b_1} + 2^{b_2} + \dots + 2^{b_k}$$

   a partir de los cuales obtenemos

   $$\tau^{-1}(x) = (a_1, \dots, a_k)$$

   donde $a_1 = b_1$ y $a_{i+1} = b_{i+1} - b_i - 1$ (para $1 \leq i < k$). $\diamond$

---

[1] Evidentemente, sin repeticiones.
[2] Nat^+ denota los números naturales exceptuando el 0.

Denotaremos el conjunto de todas las instrucciones URM por $\mathcal{I}$ y el conjunto de todos los programas URM por $\mathcal{P}$.

Teorema 11 ($\mathcal{I}$ es efectivamente denumerable)

Demostración:

Definimos la biyección $\beta : \mathcal{I} \to \mathcal{N}$ que establece una correspondencia entre los cuatro tipos de instrucciones URM a los naturales de la forma $au, 4u+1, 4u+2, 4u+3$ respectivamente. Utilizaremos a las funciones π y ς definidas en la demostración del teorema 10 anterior.

$$\begin{aligned}
\beta\big(Z(n)\big) &= 4(n-1) \\
\beta\big(S(n)\big) &= 4(n-1)+1 \\
\beta\big(T(m,n)\big) &= 4\pi(m-1,n-1)+2 \\
\beta\big(J(m,n,q)\big) &= 4\varsigma(m,n,q)+3
\end{aligned}$$

La definición de β muestra explícitamente que es computable. Para hallar β^{-1} encontremos primero los valores u, r tales que $x = 4u + r$ con $0 \le r < 4$. El valor de r indica que tipo de instrucción corresponde a $\beta^{-1}(x)$, y de u podemos encontrar efectivamente la instrucción particular de ese tipo:

si $r = 0$, entonces $\beta^{-1}(x) = Z(u+1)$,
si $r = 1$, entonces $\beta^{-1}(x) = S(u+1)$,
si $r = 2$, entonces $\beta^{-1}(x) = T\big(\pi_1(u)+1, \pi_2(u)+1\big)$,
si $r = 3$, entonces $\beta^{-1}(x) = J(m,n,q)$ donde $(m,n,q) = \varsigma^{-1}(u)$.

Por tanto, β^{-1} es efectivamente computable.
$\diamond$

Teorema 12 ($\mathcal{P}$ es efectivamente denumerable)

Demostración:

Definiremos una biyección explícita $\gamma : \mathcal{P} \to \mathcal{N}$ de la siguiente manera, utilizando las biyecciones β, τ ya mostradas: si $P = I_1, I_2, \ldots, I_s$ entonces

$$\gamma(P) = \gamma\big(\beta(I_1), \beta(I_2), \ldots \beta(I_s)\big)$$

Dado que τ y β son biyecciones, también lo es γ y γ^{-1}.
$\diamond$

La biyección γ jugará un rol preponderante en las siguientes discusiones. Para un programa P, el número $\gamma(P)$ se denomina el *número de código* de P, o el *número Gödel* de P ó simplemente el número de P. Definimos

$$\begin{aligned}
P_n &= \text{el programa con número (de código) } n \\
&= \gamma^{-1}(n)
\end{aligned}$$

y diremos que P_n es el n-ésimo programa. Por la construcción de γ, si $m \neq n$ entonces P_m difiere de P_n aunque estos programas podrían computar las mismas funciones.

El hecho que γ y γ^{-1} son efectivamente computables tiene las repercusiones siguientes:

1. Dado un programa P, podemos encontrar efectivamente el número $\gamma(P)$,

2. Dado un número n podemos encontrar efectivamente el programa $P_n = \gamma^{-1}(n)$.

Ejemplo 10 *1. Sea P el programa $T(1,3), S(4), Z(6)$. Calcularemos $\gamma(P)$:*

$$\begin{aligned}
&\beta\big(T(1,3)\big) = 4\pi(0,2) + 2 = 4\big(2^0(2\times 2+1)-1\big) + 2 = 18\\
&\beta\big(S(4)\big) = 4\times 3 + 1 = 13\\
&\beta\big(Z(6)\big) = 4\times 5 = 20
\end{aligned}$$

Por tanto

$$\begin{aligned}
\gamma(P) &= 2^{18} + 2^{18+13+1} + 2^{18+13+20+1} - 1\\
&= 2^{18} + 2^{32} + 2^{53} - 1\\
&= 9007203549970431.
\end{aligned}$$

2. Sea $n = 4127$, encontraremos P_{4127}.

$4127 = 2^5 + 2^{12} - 1$ por lo que P_{4127} es un programa con dos instrucciones I_1, I_2 donde

$$\begin{aligned}
&\beta^{-1}(I_1) = 5 = 4\times 1 + 1\\
&\beta^{-1}(I_2) = 12 - 5 - 1 = 6 = 4\times 1 + 2 = 4\pi(1,0) + 2
\end{aligned}$$

Por lo que de la definición de β, se tiene que $I_1 = S(2)$, $I_2 = T(2,1)$; y así $P_{4127} = S(2), T(2,1)$.

Por supuesto que existen otras formas posibles de definir biyecciones efectivas de $\mathcal{P}$ hacia $\mathcal{N}$; la elección presentada en γ fué arbitraria; lo que es importante recalcar es que γ y γ^{-1} son computables. Por tanto, para el resto del texto la definición de γ permanecerá fija de tal forma que para cada número particular n el significado de P_n no cambia. Por ejemplo, P_{4127} siempre denotará al programa $S(2), T(2,1)$.

Ejercicios 9 *Encontrar:*

1. $\beta\big(J(3,4,2)\big)$,

2. $\beta^{-1}(503)$,

3. el número de código del siguiente programa: $T(3,4), S(3), Z(1)$.

4. P_0, P_{100}

4.2 Enumerando Funciones Computables

Utilizando la convención de codificar (enumerar) programas, podemos enumerar funciones computables así como sus dominios y rangos. Estableceremos la notación que se seguirá en adelante.

Definición 13 (ϕ, W, E) *Para cada $a \in \mathcal{N}$ y $n \geq 1$:*

1. *$\phi_a^{(n)}$ denota la función de aridad n computada por P_a.*
   (corresponde a $f_{P_a}^{(n)}$ en la notación del capítulo 1).

2. *$W_a^{(n)}$ denota el dominio de $\phi_a^{(n)}$:*

$$W_a^{(n)} = \{(x_1, \ldots, x_n) \mid P_a(x_1, \ldots, x_n) \downarrow\}$$

3. *$E_a^{(n)}$ denota el rango de $\phi_a^{(n)}$.*

Estaremos principalmente interesados con las funciones computables unarias en los capítulos siguientes, por conveniencia se omitirá el superíndice (1): ϕ_a representa $\phi_a^{(1)}$, etc.

Ejemplo 11 *Sea $a = 4217$ del ejemplo anterior sabemos que $P_{4127} = S(2), T(2,1)$; por tanto*

$$\begin{array}{ll} \phi_{4127}(x) = 1 & \textit{para todo } x, \\ \phi_{4127}^{(n)}(x_1, x_2, \ldots, x_n) = x_2 + 1 & \textit{para } n > 1. \end{array}$$

Entonces

$$\begin{array}{ll} W_{4127} = \mathcal{N}, & E_{4127} = \{1\} \\ W_{4127}^{(n)} = \mathcal{N}^n, & E_{4127}^{(n)} = \mathcal{N}^+ \textit{ si } n > 1. \end{array}$$

Si f es computable unaria entonces existe un programa P que computa f por lo que $f = \phi_a$ donde $a = \gamma(P)$. Decimos que a es un **índice** para f. Dado que existen muchos programas distintos para computar una función dada no podemos decir que a es *el índice* de f, de hecho cada función computable tiene infinitamente muchos índices. Por tanto, cada función computable unaria aparece en la enumeración

$$\phi_0, \phi_1, \phi_2, \ldots$$

y que tal enumeración es con repeticiones. Lo anterior se aplica también a las funciones con aridad n.

Ejercicios 10 *Demostrar que toda función computable tiene infinitamente muchos índices.*

Recuérdese que denotamos la clase de todas las funciones computables de aridad n por $\mathcal{C}_n$.

Teorema 13 *$\mathcal{C}_n$ es denumerable.*

Demostración:
Utilizaremos la enumeración $\phi_0^{(n)}, \phi_1^{(n)}, \phi_2^{(n)} \ldots$ (la cual tiene repeticiones) para construír una enumeración sin repeticiones:

$$\text{Sea } \begin{cases} f(0) = 0, \\ f(m+1) = \mu z\big(\phi_z^{(n)} \neq \phi_{f(0)}^{(n)}, \phi_{f(1)}^{(n)}, \ldots, \phi_{f(m)}^{(n)}\big). \end{cases}$$

(i.e., $\phi_z^{(n)}$ difiere de cada $\phi_{f(i)}^{(n)}$, $0 \leq i \leq m$)

Entonces

$$\phi_{f(0)}^{(n)}, \phi_{f(1)}^{(n)}, \phi_{f(2)}^{(n)}, \ldots$$

es una enumeración de $\mathcal{C}_n$ sin repeticiones.

◇

Nótese que no se está asegurando que f sea computable, veremos más adelante que no es así; no obstante se puede construír (de manera bastante complicada) una función total y computable h tal que $\phi_{h(0)}^{(n)}, \phi_{h(1)}^{(n)}, \phi_{h(2)}^{(n)}, \ldots$ es una enumeración de $\mathcal{C}_n$ sin repeticiones.

Corolario 7 *$\mathcal{C}$ es denumerable.*

Demostración:

Dado que podemos expresar a $\mathcal{C}$ como $\mathcal{C} = \bigcup_{n \geq 1} \mathcal{C}_n$ entonces se sigue que la unión denumerable de conjuntos denumerables es denumerable.

Explícitamente, para cada n sea f_n la función utilizada en el teorema anterior (para dar una enumeración de $\mathcal{C}$ sin repeticiones). Sea π la biyección $\pi : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ definida en la prueba del teorema 10. Defínase $\theta : \mathcal{C} \to \mathcal{N}$ por

$$\theta\big(\phi_{f_n(m)}^{(n)}\big) = \pi\big(m, n-1\big)$$

Dado que π es biyectiva, θ también lo es.

◇

El siguiente teorema muestra que existen funciones que no son computables. La idea de su demostración es tan importante como el resultado mismo.

Teorema 14 *Existe una función unaria total que no es computable.*

Demostración:

Debemos construír una función total f que sea distinta simultáneamente de cada función que está en la enumeración $\phi_0^{(n)}, \phi_1^{(n)}, \phi_2^{(n)} \ldots$ de la clase $\mathcal{C}$:

$$f(n) = \begin{cases} \phi_n(n) + 1 & \text{si } \phi_n(n) \text{ está definida,} \\ 0 & \text{si } \phi_n(n) \text{ no está definida.} \end{cases}$$

Nótese que la construcción de f difiere de ϕ_n en el argumento n (para todo n):

- Si $\phi_n(n)$ está definida, entonces f difiere de ϕ_n en que $f(n) \neq \phi_n(n)$,
- Si $\phi_n(n)$ no está definida, entonces f difiere de ϕ_n en que $f(n)$ sí está definida.

Dado que f difiere de toda función computable ϕ_n entonces f no puede aparecer en la enumeración de $\mathcal{C}_1$ y por lo tanto no es computable. Se puede apreciar inmediatamente que f es total.

◇

4.2.1 El método de la Diagonalización de Cantor

El método que se utilizó en la construcción de la función f en la demostración del teorema anterior es un ejemplo de *diagonalización* de Cantor. La idea fundamental consiste en lo siguiente: consideremos la enumeración de las funciones $\phi_0^{(n)}, \phi_1^{(n)}, \phi_2^{(n)} \ldots$ en la siguiente tabla, en la cual las funciones se encuentran colocadas en orden creciente en la columna izquierda y los argumentos ($\mathcal{N}$) en el renglón superior

$\mathcal{C}$	0	1	2	3	4
ϕ_0	$\boxed{\phi_0(0)}$	$\phi_0(1)$	$\phi_0(2)$	$\phi_0(3)$	$\ldots$
ϕ_1	$\phi_1(0)$	$\boxed{\phi_1(1)}$	$\phi_1(2)$	$\phi_1(3)$	$\ldots$
ϕ_2	$\phi_2(0)$	$\phi_2(1)$	$\boxed{\phi_2(2)}$	$\phi_2(3)$	$\ldots$
ϕ_3	$\phi_3(0)$	$\phi_3(1)$	$\phi_3(2)$	$\boxed{\phi_3(3)}$	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Supondremos que cuando alguna $\phi_n(m)$ no está definida en la celda correspondiente se encuentra la etiqueta *'indefinida'*.

La función f se construyó tomando las entradas en la diagonal y cambiándolas sistemáticamente obtieniendo $f(0), f(1), f(2), \ldots$ de tal forma que $f(n)$ difiere de $\phi_n(n)$ para cada n. Claramente se tuvo la libertad de elegir el valor de $f(n)$ sólo se tuvo que asegurar que fuese distinto de $\phi_n(n)$. Por ejemplo, otra función total no computable es

$$g(n) = \begin{cases} \phi_n(n) + 27^n & \text{si } \phi_n(n) \text{ está definida,} \\ n^2 & \text{si } \phi_n(n) \text{ no está definida.} \end{cases}$$

Podemos resumir el método de la diagonalización (como lo utilizaremos en adelante) de la siguiente manera: supóngase que $\chi_0, \chi_1, \chi_2, \ldots$ es una enumeración de objetos del mismo tipo (funciones, números naturales, etc.). Podemos construír un objeto χ del mismo tipo que es distinto de cada χ_n utilizando el siguiente *lema*:

'Hacer que χ y χ_n difieran en n'.

Evidentemente, la interpretación del término *difieran* depende del tipo de objeto involucrado: las funciones pueden diferir en el argumento n siempre que estén definidas. En el caso de los conjuntos, la pregunta en n es si el elemento n pertenece o no al conjunto:

Ejemplo 12 *Sea $A_0, A_1, A_2, \ldots$ una enumeración de subconjuntos de $\mathcal{N}$. Podemos construír un conjunto B utilizando el lema de la diagonalización de la siguiente forma:*

$$n \in B \text{ si y sólo si } n \notin A_n$$

Claramente, para cada n, $B \neq A_n$.

Discutiremos aplicaciones muy importantes de la diagonalización en los siguientes capítulos.

Ejercicios 11 *1. Sea $f(x, y)$ total y computable. Para cada m, sea g_m la función computable dada por*

$$g_m(y) \simeq f(m, y)$$

Construya una función computable total h tal que para cada m, $h \neq g_m$.

2. *Sea $f_0, f_1, \ldots$ una enumeración de funciones parciales de $\mathcal{N} \to \mathcal{N}$. Construya una función $g : \mathcal{N} \to \mathcal{N}$ tal que $Dom(g) \neq Dom(f_i)$ para cada i.*

3. *Sea $f\mathcal{N} \to \mathcal{N}$ una función parcial, y sea $m \in \mathcal{N}$. Construya una función no computable g tal que*

$$g(x) \simeq f(x) \quad \text{para cada } x \leq m.$$

4. *Muestre que el conjunto de todas las funciones de $\mathcal{N} \to \mathcal{N}$ no es denumerable.*

5. *Muestre que el conjunto de todas las funciones totales no computables de $\mathcal{N} \to \mathcal{N}$ no es denumerable.*

4.3 El Teorema *s-m-n*

En ésta sección se demostrará un teorema que tiene usos muy importantes, especialmente en conjunción con el teorema principal del siguiente capítulo.

Primero presentaremos la idea intuitiva: sea $f(x, y)$ una función computable (no necesariamente total). Para cada valor fijo a del argumento x, f da lugar[3] a una función computable unaria g_a en donde

$$g_a(y) \simeq f(a, y)$$

Dado que g_a es computable, tiene como índice e, así que

$$f(a, y) \simeq \phi_e(y).$$

El siguiente teorema muestra que tal índice e puede obtenerse *efectivamente* a partir de a.

Teorema 15 (*s-m-n*, forma simple) *Sea $f(x, y)$ computable. Existe una función computable total $k(x)$ tal que*

$$f(x, y) \simeq \phi_{k(x)}(y)$$

[3]Church: $f(x_1, x_2) \equiv h_1\Big(\big(h_2(x_1)\big)(x_2)\Big)$: Curry.

Demostración:

Para cada a fijo, $k(a)$ será el número de código de un programa Q_a el cual, dada la configuración inicial

R_1	R_2	R_3	...
y	0	0	...

computa $f(a, y)$.

Sea F un programa que computa f. Para Q_a escribimos a F precedida por las instrucciones que transforman la configuración anterior en

R_1	R_2	R_3	...
y	y	0	...

Q_a es

$$\begin{array}{ll} & T(1,2) \\ & Z(1) \\ a \text{ veces} & \left\{ \begin{array}{l} S(1) \\ \vdots \\ S(1) \end{array} \right. \\ & F \end{array}$$

Defínase

$$k(a) = \text{ número de código de } Q_a.$$

Dado que F es fijo y a partir del hecho que la enumeración γ de los programas es efectiva, por la tesis de Church k es computable. Así

$$\phi_{k(a)}(y) \simeq f(a, y)$$

para cada a.

◇

Al teorema *s-m-n* a menudo se le denomina *teorema de Parametrización* dado que muestra que un índice para una función computable se puede encontrar efectivamente a partir de un parámetro (a) del cual depende. A menudo, a la función k que sirve como índice le denominaremos *función indicadora.*

El indexado se puede extender a los dominios y rangos de las funciones computables:

Ejemplo 13

1. *Sea $f(x, y) = y^x$, por el teorema anterior existe una función indicadora total y computable k tal que $\phi_{k(x)}(y) = y^x$. Para cada n fijo, $k(n)$ es un índice para y^n.*

2. *Sea* $f(x, y) = \left\{ \begin{array}{ll} y & \textit{si } y \textit{ es múltiplo de } x \\ \textit{indefinida} & \textit{de lo contrario.} \end{array} \right.$

   Dado que f es computable, existe k tal que $\phi_{k(x)}(y) \simeq f(x, y)$. Por tanto, para cada n fijo

$$\begin{array}{lll} \phi_{k(x)}(y) \textit{ está definida} & \textit{si y sólo si} & y \textit{ es múltiplo de } n \\ & \textit{si y sólo si} & y \in \textit{Rango}\big(\phi_{k(x)}(y)\big). \end{array}$$

lo cual significa que

$$\begin{aligned} W_{k(n)} &= n\mathcal{N} \text{ (i.e., el conjunto de todos los múltiplos de } n\text{)} \\ &= E_{k(n)}. \end{aligned}$$

Por tanto se dispone de un indexado efectivo de la secuencia de conjuntos $(n\mathcal{N})$ *como*

(a) los dominios de funciones computables,

(b) los rangos de funciones computables.

La manera inmediata para generalizar el teorema anterior es reemplazar las variables individuales x, y por tuplas de tamaño m y n respectivamente. Debe reconocerse que la función k depende efectivamente de un programa particular para la función original f, por lo que en vez de considerar una función computable fija $f(x, y)$ consideramos una función computable general $\phi_e^{(m+n)}(\vec{x}, \vec{y})$ de tal suerte que se indica la búsqueda efectiva de un z tal que para cada e y $\vec{x}$

$$\phi_e^{(m+n)}(\vec{x}, \vec{y}) \simeq \phi_z^{(n)}(y)$$

Teorema 16 (El Teorema s-m-n**)** *Para cada* $m, n \geq 1$ *existe una función computable total* $s_n^m(e, \vec{x})$ *(de aridad* $(m+1)$*) tal que*

$$\phi_e^{(m+n)}(\vec{x}, \vec{y}) \simeq \phi_{s_n^m(e,\vec{x})}^{(n)}(y)$$

Demostración:

Generalizaremos la demostración del teorema anterior (forma simple).

Para cada $i \geq 1$ sea $Q(i, x)$ la subrutina

$$Z(i), \quad \underbrace{S(i), \ldots, S(i)}_{x \text{ veces}}$$

que reemplaza los contenidos actuales de cada R_i por x. Para m, n fijos, se define $s_n^m(e, \vec{x})$ como el número de código del siguiente programa:

$$\left.\begin{array}{l} T(n, m+1) \\ \vdots \\ T(2, m+2) \\ T(1, m+1) \\ Q(1, x_1) \\ Q(2, x_2) \\ \vdots \\ Q(m, x_m) \\ P_e \end{array}\right\}$$

transforma cualquier configuración

R_1	...	R_n	...
y_1	...	y_n	...

en:

R_1	...	R_m	R_{m+1}	... R_{m+n}	...	
x_1	...	x_m	y_1	...	y_n	...

A partir de éste definición explícita y de la efectividad de γ y γ^{-1} se concluye que s_n^m es efectivamente computable por la tesis de Church.

$\diamond$

No es difícil comprobar que la función s_n^m es primitiva recursiva: para cada m existe una función s^m (también primitiva recursiva) que satisface el teorema para toda n.

Ejercicios 12 *1. Mostrar que existe una función computable total k tal que para cada n, $k(n)$ es un índice para la función $[\sqrt[n]{x}]$.*

2. Demostrar que existe una función computable total k tal que para cada n, $W_{k(n)}$ es el conjunto de las potencias de n.

3. Sea $n \geq 1$. Demostrar que existe una función computable total s

$$W^(n)_{s(x)} = \{(y_1, \ldots, y_n) \mid y_1 + y_2 + \ldots + y_n = x\}$$

4. Demostrar que las funciones s_n^m definidas en el teorema anterior son todas primitivas recursivas.

5. Demostrar que para cada m existe una función total computable s^m de aridad $(m+1)$ tal que para cada n

$$\phi_e^{(m+n)}(\vec{x}, \vec{y}) \simeq \phi_{s^m(e,\vec{x})}^{(n)}(y)$$

donde $\vec{x}$ y $\vec{y}$ son tuplas de tamaño m y n respectivamente.

Capítulo 5

Programas Universales

En éste capítulo se discutirá el resultado que afirma que existen *programas universales*: programas que en cierto sentido incorporan a todos los demás programas. Éste resultado es uno de los pilares de la teoría de computabilidad (junto con el teorema *s-m-n*).

Entre las aplicaciones importantes de los programas universales se tiene la construcción de funciones no computables específicas y de predicados indecidibles. A su vez, se muestra el uso de un programa universal para construír una función computable total que no es primitiva recursiva.

Al final de éste capítulo se discuten usos del teorema *s-m-n* junto con los programas universales para mostrar que ciertas operaciones sobre índices de funciones computables son efectivas.

5.1 Programas Universales y Funciones Universales

Considérese a la función $\psi(x, y)$ definida como

$$\psi(x, y) \simeq \phi_x(y)$$

La función ψ abarca a toda $\phi_0, \phi_1, \phi_2, \ldots$ dado que para cualquier m, la función g

$$g(y) \simeq \psi(m, y)$$

es otro nombre para ϕ_m. Decimos que ψ es la *función universal* para las funciones computables unarias.

Definición 14 *La función universal para funciones computables n-arias es la función $\psi_U^{(n)}$ de aridad $(n+1)$ dada por*

$$\psi_U^{(n)}(e, x_1, \ldots, x_n) \simeq \phi_e^{(n)}(e, x_1, \ldots, x_n)$$

Escribimos ψ_U oara denotar $\psi_U^{(1)}$.

¿Es en general $\psi_U^{(n)}$ computable? Si la respuesta es afirmativa entonces cualquier programa P que compute ψ_U incorporaría todos los demás programas y P se le podría denominar con justa razón *programa universal*. El punto importante, sin embargo, es que un programa universal P no necesita contener a todos los demás programas P_e sino que sólo necesita tener la habilidad de *decodificar* cualquier número e y por tanto *imitar* a P_e[1].

Teorema 17 *Para cada n, la función universal $\psi_U^{(n)}$ es computable.*

Demostración:

Primero daremos una justificación (prueba informal) apelando la tesis de Church y entonces bosquejaremos una prueba formal.

Sea un n fijo, nos dan un índice e y la tupla $\vec{x}$ (de tamaño n). Podemos describir un procedimiento informal para computar $\psi_U^{(n)}(e, \vec{x})$ de la siguiente forma:

'Decodificar el número e y escribir el programa P_e. Entonces imitar el cómputo de $P_e(\vec{x})$ paso a paso, en cada paso se debe describir la configuración de los registros y la siguiente instrucción a ejecutarse. Cuando el cómputo de detenga, el valor requerido de $\psi_U^{(n)}(e, \vec{x})$ estará en R_1'.

Podríamos concluír inmediatamente por la tesis de Church que $\psi_U^{(n)}(e, \vec{x})$ es computable. Sin embargo, a continuación se bosquejará una prueba formal, cuyo plan consiste en mostrar primero cómo utilizar un sólo número σ para codificar la situación actual durante el cómputo; entonces mostrar que existe una función computable que expresa la dependencia de σ en

1. el número de (código del) programa e,
2. la entrada $\vec{x}$,
3. el número de pasos del cómputo que han sido completados.

Veremos que ésto será suficiente para demostrar el teorema.

Recordemos que la situación actual de un cómputo para el programa $P_e(\vec{x})$ se especifica completamente por la configuración actual de los registros $r_1, r_2, \ldots$ y el número j de la siguiente instrucción a ejecutarse. Gracias a que solamente un número grande (pero finito) de los r_i's no son cero, podemos especificar la configuración actual por un sólo número[2]:

$$c = 2^{r_1} \times 3^{r_2} \times \ldots = \prod_{i \geq 1} p_i^{r_i}$$

A éste número le llamaremos el *código de la configuración* ó simplemente la *configuración*. Podemos recuperar los contenidos r_i de cada R_i mediante $r_i = (c)_i$ (utilizando la función del teorema 8 (4), pág. 25).

[1]Intuitivamente, el programa universal hace las veces de *intérprete* que ejecuta a su vez otros programas.

[2]Recuérdese que p_i es el i-ésimo número primo.

Podemos codificar la descripción completa de la situación actual mediante el número $\sigma = \pi(c, j)$, el cual lo llamaremos el *estado* actual del cómputo $P_e(\vec{x})$. Seguiremos la convención de que si el cómputo se ha detenido, entonces $j = 0$ y c es la configuración final. Nótese que $c = \pi_1(\sigma)$ y $j = \pi_2(\sigma)$ (donde π_1, π_2 son las funciones computables definidas en el capítulo anterior, teorema 10, pág. 38).

Sabemos que los valores de c, j y σ cambian durante el transcurso del cómputo, podemos expresar su dependencia con respecto del número de programa e, la entrada $\vec{x}$ y el número de pasos completados mediante las siguientes funciones (de aridad $(n+2)$):

1. $c_n(e, \vec{x}, t)$ = la configuración después de que t pasos de $P_e(\vec{x})$ han sido completados, (será la configuración final si $P_e(\vec{x}) \downarrow$ en t pasos o menos).

2. $$j_n(e, \vec{x}, t) = \begin{cases} \text{el número de la siguiente} & \text{si } P_e(\vec{x}) \text{ no} \\ \text{instrucción de } P_e(\vec{x}) & \text{se ha detenido después} \\ \text{tras completarse } t \text{ pasos,} & \text{de } t \text{ pasos o menos} \\ & \\ 0 & \text{si } P_e(\vec{x}) \downarrow \text{en } t \text{ pasos o menos} \end{cases}$$

3. $\sigma_n(e, \vec{x}, t)$ = el estado del cómputo $P_e(\vec{x})$ después de t pasos, $= \pi\big(c_n(e, \vec{x}, t), j_n(e, \vec{x}, t)\big)$.

El propósito es demostrar que σ_n (y en consecuencia c_n y j_n) son computables[3]. Para apreciar por qué es ese el propósito, supóngase que ya hemos demostrado que σ_n, c_n y j_n son computables. Ello significa que si el cómputo $P_e(\vec{x})$ se detuvo lo hizo en un número de pasos dado por $\mu t\big(j_n(e, \vec{x}, t) = 0\big)$; la configuración final queda $c_n\big(e, \vec{x}, \mu t(j_n(e, \vec{x}, t) = 0)\big)$, por lo que podemos expresar el comportamiento del programa universal $\psi_U^{(n)}$ en términos de la expresión anterior

$$\psi_U^{(n)}(e, \vec{x}) \simeq \Big(c_n\big(e, \vec{x}, \mu t(j_n(e, \vec{x}, t) = 0)\big)\Big)_1$$

De esa forma, si c_n y j_n son computables[4] también lo es $\psi_U^{(n)}$, de tal suerte que la prueba principal queda completada.

Partimos del supuesto que σ_n, c_n y j_n fuesen computables, por lo que podemos aplicar la tesis de Church para mostrarlo: tenemos el siguiente algoritmo informal para obtener $\sigma_n(e, \vec{x}, t+1)$ efectivamente a partir de e y $\sigma_n(e, \vec{x}, t)$:

'Decodificar $\sigma_n(e, \vec{x}, t)$ para encontrar los números $c = c_n(e, \vec{x}, t)$ y $j = j_n(e, \vec{x}, t)$. Si $j = 0$ entonces $\sigma_n(e, \vec{x}, t+1) = \sigma_n(e, \vec{x}, t)$. De lo contrario, describir la configuración codificada por c, típicamente tendría la siguiente forma:

$(c)_1$	$(c)_2$	$\ldots$	$(c)_m$	0	$\ldots$

y decodificar e para obtener el programa P_e. Encontrar la j-ésima instrucción en P_e y operarla con la configuración anterior produciendo una nueva configuración

[3] Nótese que estamos describiendo el comportamiento del programa P_e en términos de funciones computables.

[4] De hecho, primitivas recursivas.

con código c'. Encontrar también el número de la siguiente instrucción j' ($j' = 0$ si el cómputo ha terminado). Tenemos entonces que

$$\sigma_n(e, \vec{x}, t+1) = \pi(c', j')$$

lo cual muestra informalmente que $\sigma_n(e, \vec{x}, t)$ es computable por recursión en t dado que para $t = 0$ se tiene

$$\sigma_n(e, \vec{x}, 0) = \pi(2^{x_1} \times 3^{x_2} \times \ldots \times p_n^{x_n}, 1)$$

Por la tesis de Church, σ_n es computable.
$\diamond$

De la prueba anterior se obtiene que

Corolario 8 *Para cada $n \geq 1$, los siguientes predicados son decidibles:*

1. $S_n(e, \vec{x}, y, t) \equiv$ *'$P_e(\vec{x}) \downarrow y$ en t o menos pasos',*
2. $H_n(e, \vec{x}, t) \equiv$ *'$P_e(\vec{x}) \downarrow$ en t o menos pasos'.*

Demostración:

1. $S_n(e, \vec{x}, y, t) \equiv$ '$j_n(e, \vec{x}, t) = 0$ y $\big(c_n(e, \vec{x}, t)\big)_1 = y$'
2. $H_n(e, \vec{x}, t) \equiv$ '$j_n(e, \vec{x}, t) = 0$'.

$\diamond$

Corolario 9 (Forma normal de Kleene) *Existe una función computable total $\mathbf{U}(x)$ y un predicado decidible $T_n(e, \vec{x}, z)$ (para cada $n \geq 1$) tal que*

1. $\phi_e^{(n)}(\vec{x})$ *está definida si y sólo si* $\exists z T_n(e, \vec{x}, z)$,
2. $\phi_e^{(n)}(\vec{x}) \simeq \mathbf{U}\big(\mu z T_n(e, \vec{x}, z)\big)$.

Demostración:

Para encontrar si $\phi_e^{(n)}(\vec{x})$ está definida y el valor de arguento en el cual está definida necesitamos hallar una pareja de números y, t tales que $S_n(e, \vec{x}, y, t)$. Recordemos que el operador μ permite indicar búsqueda efectiva de un número que tenga una propiedad dada: para ello codificamos la pareja $(z)_1, (z)_2$ en el número z. Conforme z varía sobre $\mathcal{N}$, la pareja $\big((z)_1, (z)_2\big)$ varía sobre $\mathcal{N} \times \mathcal{N}$. Definimos

$$T_n(e, \vec{x}, z) \equiv S_n\big(e, \vec{x}, (z)_1, (z)_2\big)$$

Para *1.*, supóngase que $\phi_e^{(n)}(\vec{x})$ está definida, por lo tanto existen y, t tales que $S_n(e, \vec{x}, y, t)$ está definida, asignando $z = 2^y \times 3^t$ obtenemos el valor de $T_n(e, \vec{x}, z)$. A su vez, si existe un z tal que $T_n(e, \vec{x}, z)$ esté definida, entonces de la definición de T_n, el programa converge: $P_e(\vec{x}) \downarrow$; lo cual significa que $\phi_e^{(n)}(\vec{x})$ está definida.

Para *2.* se puede apreciar de la definición de T_n que $\phi_e^{(n)}(\vec{x})$ está definida, entonces apra cualquier z tal que $T_n(e, \vec{x}, z)$ esté definida, se tiene que $\phi_e^{(n)}(\vec{x}) = (z)_1$. Por lo tanto, podemos hacer $\mathbf{U}(z) = (z)_1$ y entonces

$$\phi_e^{(n)}(\vec{x}) \simeq \mathbf{U}\big(\mu z T_n(e, \vec{x}, z)\big)$$

◇

La forma normal de Kleene indica que toda función computable (incluso recursiva parcial) puede obtenerse a partir de funciones recursivas primitivas utilizando a lo más una aplicación del operador μ, ofreciendo un método para ello.

La técnica de buscar parejas de números considerando en un sólo número z para codificar la pareja $(z)_1, (z_)2$ se usa muy frecuentemente en teoría de computabilidad. Tal técnica puede extenderse para buscar secuencias $(x_1, x_2, \ldots, x_n)$ para cualquier $n > 1$.

Ejercicios 13 *1. Mostrar que existe un predicado decidible $Q(x, y, z)$ tal que:*

(a) $y \in E_x$ si y sólo si $\exists z Q(x, y, z)$

(b) si $y \in E_x$ y $Q(x, y, z)$ entonces $\phi_x\big((z)_1\big) = y$.

2. Deducir que existe una función computable $g(x, y)$ tal que:

(a) $g(x, y)$ está definida si y sólo si $y \in E_x$

(b) si $y \in E_x$ entonces $g(x, y) \in W_x$ y $\phi_x\big(g(x, y)\big) = y$, i.e. $g(x, y) \in \phi_x^{-1}\big(\{y\}\big)$.

3. Deducir que si f es computable inyectiva (no necesariamente total o sobreyectiva) entonces f^{-1} es computable.

4. Sean f, g computables, asúmase que T_1 es decidible, demostrar que la función

$$h(x) = \begin{cases} 1 & \textit{si } x \in \textit{Dom}(f) \textit{ o } x \in \textit{Dom}(g), \\ \textit{indefinida} & \textit{de lo contrario.} \end{cases}$$

es computable.

5.2 Dos aplicaciones del Programa Universal

Ilustraremos ahora el uso de las funciones universales en las construcciones por diagonalización.

Teorema 18 *El problema 'ϕ_x es total' (para cualquier x) es indecidible.*

Demostración:

Sea g la función característica del problema,

$$g(x) = \begin{cases} 1 & \text{si } \phi_x \text{ es total,} \\ 0 & \text{de lo contrario.} \end{cases}$$

para demostrar que g no es computable utilizamos diagonalización para construír una función f que es distinta de cada función computable de tal suerte que si g es computable entonces f también lo es:

$$f(x) = \begin{cases} \phi_x(x) + 1 & \text{si } \phi_x \text{ es total,} \\ 0 & \text{de lo contrario.} \end{cases}$$

Claramente, f es total y difiere de toda función computable ϕ_x. Utilizando g y ψ_U podemos escribir a f como

$$f(x) = \begin{cases} \psi_U(x,x) + 1 & \text{si } g(x) = 1, \\ 0 & \text{si } g(x) = 0. \end{cases}$$

Si suponemos que g es computable dado que ψ_U lo es, por la tesis de Church f es computable, lo cual es una contradicción. Por tanto, g no es computable.
$\diamond$

Teorema 19 *Existe una función computable total que no es primitiva recursiva.*

Demostración:

Primero, una prueba informal[5]: cada función primitiva recursiva puede especificarse por un *plan* que indica las funciones básicas utilizadas en la secuencia de operaciones ejecutadas en su construcción. Adoptaremos la siguiente notación para describir el plan:

- $Sub(f; g_1, g_2, \ldots, g_m)$ denota la función obtenida aplicando substitución de $g_1, g_2, \ldots, g_m$ en f, $(f : \mathcal{N}^m \to \mathcal{N}, g_i : \mathcal{N}^n \to \mathcal{N})$
- $Rec(f, g)$ denota la función obtenida a partir de f y g por recursión $(f : \mathcal{N}^n \to \mathcal{N}, g : \mathcal{N}^{n+2} \to \mathcal{N})$

Si a la función sucesor la describimos por $x + 1$ tenemos por ejemplo el siguiente plan para la función $f(x) = x^2$ (cada g_i indica una función intermedia):

[5]Recordemos que las funciones primitivas recursivas son aquellas que se pueden construír a partir de las funciones básicas mediante una secuencia de aplicaciones de las operaciones de substitución y recursión.

Paso:	Plan	Explicación
1	$g_1 = Sub(S; U_3^3)$	$g_1(x,y,z) = U_3^3(x,y,z) + 1 = z + 1$
2	$g_2 = Rec(U_1^1, g_1)$	$\begin{cases} g_2(x,0) & = U_1^1(x) = x, \\ g_2(x,y+1) & = g_1\big(x,y,g_2(x,y)\big) \\ & = g_2(x,y) + 1 \end{cases}$ Así $g_2(x,y) = x + y$.
3	$g_3 = Sub(g_2; U_1^3, U_3^3)$	$g_3(x,y,z) = g_2(x,z) = x + z$.
4	$g_4 = Rec(\mathbf{0}, g_3)$	$\begin{cases} g_4(x,0) & = 0, \\ g_4(x,y+1) & = g_3\big(x,y,g_4(x,y)\big) \\ & = x + g_4(x,y) \end{cases}$ Así $g_4(x,y) = x \times y$.
5	$f = Sub(g_4; U_1^1, U_1^1)$	$f(x) = g_4(x,x) = x^2$.

Por tanto un plan es como un programa en el sentido que es una especificación finita, explícita y utiliza funciones primitivas recursivas de una función computable.

De igual forma que con los programas, podemos enumerar los planes para funciones primitivas recursivas unarias, de tal forma que podemos definir

$$\theta_n = \text{la función primitiva recursiva unaria definida por el plan } n$$

Entonces $\theta_0, \theta_1, \ldots$ es una enumeración efectiva de todas las funciones primitivas recursivas.

Como toda función primitiva recursiva es computable existe una función total $\mathbf{p}$ tal que para cada n, $\mathbf{p}$ es el número del programa (plan) que computa θ_n,

$$\theta_n = \phi_{\mathbf{p}(n)}$$

Ahora el punto crucial es que podamos encontrar tal función $\mathbf{p}$ que sea computable, lo cual lo podemos asumir informalmente por la tesis de Church: de los teoremas 1 y 3 en el cap. anterior se mostró explícitamente cómo construír un programa para las funciones $Sub(f; g_1, \ldots, g_m)$ y $Rec(f,g)$ respectivamente. Por tanto, dado un plan para una función primitiva recursiva f que involucra funciones intermedias $g_1, \ldots, g_k$ podemos encontrar efectivamente programas para tales funciones y por tanto para f. Ello significa que existe una función computable $\mathbf{p}$ tal que $\theta_n = \phi_{\mathbf{p}(n)}$, lo cual, por la tesis de Church, significa a su vez que $\mathbf{p}$ es computable.

Apartir de $\mathbf{p}$ y la función universal ψ_U podemos definir una función total computable g que difiere de cada función primitiva recursiva θ_n, utilizando diagonalización:

$$\begin{aligned} g(x) &= \theta_x(x) + 1 \\ &= \phi_{\mathbf{p}(x)}(x) + 1 \\ &= \psi_U\big(\mathbf{p}(x), x\big) + 1. \end{aligned}$$

De aquí, podemos apreciar que g es total pero no es primitiva recursiva (pues difiere de cada θ_x en x), pero es computable.

$\diamond$

5.3 Operaciones efectivas sobre funciones computables

Ilustraremos otra aplicación importante de las funciones universales en conjunción con el teorema *s-m-n*.

Considérense las siguientes operaciones sobre funciones computables y sus dominios:

1. Combinar ϕ_x y ϕ_y para formar el producto $\phi_x \phi_y$,
2. Formar la unión $W_x \cup W_y$ de W_x, W_y.

Nos preguntamos inmediatamente si podemos realizar tales operaciones de manera efectiva. Dado que en gran medida tales operaciones involucran clases de objetos infinitos (funciones o conjuntos), parecen por tanto escapar de la noción informal de computabilidad (la cual implícitamente se aplica solamente a clases de objetos finitos). No obstante, veremos por ejemplo que un índice para la función $\phi_x \phi_y$ se puede obtener de *manera efectiva a partir de los índices* x, y.

Ejemplo 14

1. *Existe una función computable total $s(x,y)$ tal que para toda x, y, $\phi_{s(x,y)} = \phi_x \phi_y$.*

   Demostración:

   Sea $\begin{array}{ll} f(x,y,z) & \simeq \phi_x(z)\phi_y(z) \\ & \simeq \psi_U(x,z)\psi_U(y,z) \end{array}$.

   f es computable, por el teorema s-m-n existe una función total computable $s(x,y)$ tal que $f(x,y,z) \simeq \phi_{s(x,y)}(z)$; por tanto $\phi_{s(x,y)} = \phi_x \phi_y$.

2. *Sea $g(x) = s(x,x)$ (con s la función anterior), tenemos que $\left(\phi_x\right)^2 = \phi_{g(x)}$.*

3. *Existe una función computable total $s(x,y)$ tal que $W_{s(x,y)} = W_x \cup W_y$.*

   Demostración:

   Sea $f(x,y,z) = \begin{cases} 1 & si\ z \in W_x\ o\ z \in W_y, \\ indefinida & en\ otro\ caso. \end{cases}$.

   Por la tesis de Church y la computabilidad de ψ_U, f es computable: existe una función computable total $s(x,y)$ tal que $f(x,y,z) \simeq \phi_{s(x,y)}(z)$. Claramente $W_{s(x,y)} = W_x \cup W_y$.

4. **Efectividad de inversas:** *Sea $g(x,y)$ computable:*

   (a) $g(x,y)$ está definida si y sólo si $y \in E_x$,

   (b) si $y \in E_x$ entonces $g(x,y) \in W_x$ y $\phi_x\big(g(x,y)\big) = y$

   Por el teorema s-m-n existe una función total computable k tal que $g(x,y) \simeq \phi_{k(x)}(y)$. Apartir de las anteriores condiciones tenemos

   (a) $W_{k(x)} = E_x$,

(b) $E_{k(x)} \subseteq W_x$,
si $y \in E_x$ *entonces* $\phi_x\big(\phi_{k(x)}(y)\big) = y$.

Por tanto, si ϕ_x *es inyectiva, entonces* $\phi_{k(x)} = \phi^{-1}$ *y* $E_{k(x)} = W_x$.

5. **Efectividad de la recusión:** *sea* $\vec{x}$ *una* n*-ada y considérese la función* f*:*

$$f(e_1, e_2, \vec{x}, 0) \simeq \phi_{e_1}^{(n)}(\vec{x}),$$
$$f(e_1, e_2, \vec{x}, y+1) \simeq \phi_{e_2}^{(n+2)}\big(\vec{x}, y, f(e_1, e_2, \vec{x}, y)\big)$$

Utilizando las funciones universales $\psi_U^{(n)}$ *y* $\psi_U^{(n+2)}$ *para reescribir las expresiones de la derecha,* f *es computable. Para* e_1, e_2 *fijos, la función* $g(\vec{x}, y) \simeq f(e_1, e_2, \vec{x}, y)$ *se obtiene de* $\phi_{e_1}^{(n)}$ *y* $\phi_{e_2}^{(n+2)}$ *por recursión. Por el teorema s-m-n existe una función computable total* $r(e_1, e_2)$ *tal que*

$$\phi_{r(e_1,e_2)}^{(n+1)}(\vec{x}, y) \simeq f\big(e_1, e_2, \vec{x}, y\big)$$

Por tanto, $r(e_1, e_2)$ *es un índice para la función de aridad* $(n+1)$ *obtenida de* $\phi_{e_1}^{(n)}$ *y* $\phi_{e_2}^{(n+2)}$ *por recursión:*

$$\phi_{r(e_1,e_2)}^{(n+1)} = Rec\big(\phi_{e_1}^{(n)}, \phi_{e_2}^{(n+2)}\big)$$

para cualquier e_1, e_2.

La moraleja es que podemos referirnos indistintamente de una función computable por un índice y a la vez utilizar una función universal indicada apropiadamente. Las operaciones entre funciones computables se reducen a operaciones sobre índices. Nótese que con los índices podemos hacer *autoreferencia:* ϕ_x hace referencia a ella misma.

Los siguientes ejercicios dan más ejemplos del uso del teorema *s-m-n* para demostrar que las operaciones que se describen son efectivas sobre índices

Ejercicios 14 *1. Mostrar que existe una función total computable* $k(e)$ *tal que para todo* e*, si* ϕ_e *es la función característica para un predicado decidible* $M(x)$*, entonces* $\phi_{k(e)}$ *es la función característica para 'no* $M(x)$*'.*

2. *Mostrar que existe una función total computable* $k(x)$ *tal que para cada* x, $E_{k(x)} = W_x$.

3. *Mostrar que existe una función total computable* $s(x, y)$ *tal que para toda* x, y, $E_{s(x,y)} = E_x \cup E_y$.

4. *Supóngase que* $f(x)$ *es computable, mostrar que existe una función computable total* $k(x)$ *tal que para toda* x, $W_{k(x)} = f^{-1}\big(W_x\big)$.

5. *Demostrar el equivalente del ejemplo 5 anterior para las operaciones de substitución y minimización, i.e.,*

(a) Para $m, n \geq 1$ fijos, existen una función total computable $s(e, e_1, \ldots, e_m)$ tal que

$$\phi^{(n)}_{s(e,e_1,\ldots,e_m)} = Sub(\phi^{(m)}_e, \phi^{(m)}_{e_1}, \phi^{(m)}_{e_2}, \ldots, \phi^{(m)}_{e_m})$$

(b) Para $n \geq 1$ fijo, existen una función total computable $k(e)$ tal que para toda e

$$\phi^{(n)}_{k(e)}(\vec{x}) \simeq \mu y\big(\phi^{(n+1)}_e(\vec{x}, y) = 0\big)$$

(Podemos extender la notación para expresar $\phi^{(n)}_{k(e)} = Min(\phi^{(n+1)}_e)$).

Capítulo 6

Decidibilidad, Indecidibilidad y Decidibilidad Parcial

6.1 Problemas indecidibles en la computabilidad

Teorema 20 (El problema de Paro)

6.2 Lógica Matemática

Teorema 21 (La Validez en Primer Orden es Indecidible)

6.3 Predicados Parcialmente Decidibles

Capítulo 7

Conjuntos Recursivos y Recursivamente Enumerables

7.1 Conjuntos Recursivos

7.2 Conjuntos Recursivamente Enumerables

7.3 Conjuntos Productivos y Creativos

7.4 Conjuntos Simples

Capítulo 8

Aritmética y el Teorema de Incompletitud de Gödel

8.1 Aritmética Formal

8.2 Incompletitud

Teorema 22 (Teorema de Incompletitud de Gödel simplificado)

8.3 Teorema de Incompletitud de Gödel

8.4 Indecidibilidad

Capítulo 9

Reducidibilidad y Grados

9.1 Reducidibilidad muchos-a-uno

9.2 Grados de Reducidibilidad

9.3 Conjuntos Recursivamente Enumerables *m-completos*

9.4 Computabilidad Relativa

9.4.1 Otros enfoques a la Computabilidad Relativa

9.5 Reducidibilidad Turing