Teoría de la Computabilidad
FCC BUAP

José de Jesús Lavalle Martínez

Índice general

Resumen

Este documento es una traducción de partes del libro *Computabiity Theory - An Introduction to Recursion Theory* de Herbert B. Enderton.

Capítulo 1

El concepto de computabilidad

1.1. El concepto informal

1.1.1. Conjuntos decidibles

La teoría de la computabilidad, también conocida como teoría de la recursión, es el área de las matemáticas que trata con el concepto de *procedimiento efectivo*, un procedimiento que se puede llevar a cabo siguiendo reglas específicas. Por ejemplo, podríamos preguntar si existe algún procedimiento efectivo -algún algoritmo- que, dada una sentencia sobre los enteros, decidirá si la sentencia es verdadera o falsa. En otras palabras, ¿es el conjunto de sentencias verdaderas sobre los enteros *decidible*? (Veremos posteriormente que la respuesta es negativa.)

O un ejemplo más simple, el conjunto de números primos es ciertamente un conjunto decidible. Esto es, existen procedimientos muy mecánicos, los cuales se enseñan en las escuelas, para decidir si cualquier número entero dado es o no un número primo. (Para un número muy grande, el procedimiento enseñado en las escuelas prodría tardar mucho tiempo.) Si queremos, podemos ejecutar un programa de computadora para ejecutar el procedimiento. Aún más simple, el conjunto de enteros pares es decidible. Podemos escribir un programa de computadora que, dado un entero, muy rápidamente decidirá si es o no par. Nuestro objetivo es estudiar qué problemas de decisión se pueden resolver (en principio) mediante un programa de computadora, y qué problemas de decisión (si hay) no se pueden resolver.

De manera más general, considere un conjunto S de números naturales. (Los números naturales son 0, 1, 2, ... En particular 0 es natural.) Decimos

que S es un conjunto *decidible* si existe un procedimiento efectivo que, dado cualquier número natural, eventualmente terminará dándonos la respuesta "Sí"si el número dado es un miembro de S y "No"si no es un miembro de S.

(Inicialmente, vamos a examinar la computabilidad en el contexto de números naturales. Posteriormente, veremos que los conceptos de computabilidad pueden ser transferidos fácilmente al contexto de cadenas de letras de un alfabeto finito. En ese contexto, podemos considerar un conjunto S de cadenas, tal como el conjunto de ecuaciones, como $x(y+z) = xy+xz$), que se cumplen en el álgebra de números reales. Pero para empezar, consideraremos conjuntos de números naturales.)

Y por un *procedimiento efectivo* entenderemos un procedimiento mediante el cual podemos dar instrucciones exactas -un programa- para llevar a cabo el procedimiento. Seguir estas instrucciones no debe demandar ideas brillantes de parte del agente (humano o máquina). Debe ser posible, al menos en principio, hacer las instrucciones tan explícitas que puedan ser ejecutadas por un empleado diligente (quien es muy bueno para seguir instrucciones pero no es muy ingenioso) o una máquina (la cual no piensa en absoluto).

Esto es, debe ser posible para nuestras instrucciones ser *implementadas mecánicamente*. (Uno puede imaginar a un matemático tan brillante que pueda ver una sentencia de la aritmética y decir si es verdadera o falsa. Pero no puede pedirle al oficinista que haga ésto. Y no exite un programa de computadora para hacer ésto. No es solamente que no hemos tenido éxito al escribir tal programa. ¡Realmente podemos probar que no es posible que exista tal programa!)

No obstante estas instrucciones deben, por supuesto, ser de longitud finita, no imponemos alguna cota superior a su posible longitud. No descartamos la posibilidad de que el número de instrucciones pueda ser absurdamente grande. (Si el número de instrucciones excede al número de electrones en el universo, solamente nos encojeremos de hombros y decimos, "Este es un programa muy grande".) Insistimos sólo en que las instrucciones -el programa- sean de longitud finita, para que podamos comunicarselas a la persona o máquina que haga los cálculos. (No hay manera de darle a alguien todo un objeto infinito.)

Similarmente, para obtener los conceptos más amplios, no imponemos cotas sobre el tiempo que el procedimiento pueda consumir antes de que nos dé la respuesta. Tampoco imponemos una cota sobre la cantidad de espacio de almacenamiento (papel de borrador) que el procedimiento pudiera necesitar. (El procedimiento puede, por ejemplo, necesitar utilizar números

muy grandes que requieren una cantidad sustancial de espacio simplemente para escribirlos.) Sólamente insistiremos en que el procedimiento nos dé la respuesta eventualmente, en cierta extensión finita de tiempo. Lo que está excluido definitivamente es hacer infinitos pasos y *luego* dar la respuesta.

En el capítulo 7, consideraremos conceptos más restrictivos, donde la cantidad de tiempo es limitada de alguna manera, así como excluiremos la posibilidad de tiempos de ejecución ridículamente grandes. Pero inicialmente, queremos evitar tales restricciones para obtener el caso límite, donde limitaciones prácticas sobre el tiempo de ejecución o espacio de memoria son eliminadas. Es bien conocido que en el mundo real la rapidez y capacidad de las computadoras ha estado creciendo sostenidamente. Queremos ignorar la rapidez y capacidad reales, en su lugar queremos preguntar cuáles son los límites puramente teóricos.

La descripción precedente de procedimiento efectivo es ciertamente vaga e imprecisa. En la siguiente sección, veremos como se puede hacer precisa esta descripción vaga, como el concepto se puede hacer un concepto *matemático*. No obstante, la idea informal de lo que se puede hacer mediante un procedimiento efectivo, esto es, lo que es *calculable*, puede ser muy útil. El rigor y la precisión pueden esperar hasta el *siguiente* capítulo. Primero necesitamos un sentido de hacia dónde vamos.

Por ejemplo, cualquier conjunto finito de números naturales debe ser decidible. El programa para el procedimiento de decisión puede incluir simplemente una lista de todos los números en el conjunto. Luego dado un número, el programa puede checarlo contra la lista. Así, el concepto de decidibilidad sólo es interesante para conjuntos infinitos.

Nuestra descripción de procedimientos efectivos, vaga como es, ya muestra que tan limitante es el concepto de decidibilidad. Uno puede, por ejemplo, utilizar los conceptos de conjuntos contables e incontables (ver el apéndice para un resumen de estos conceptos). No es difícil ver que la cantidad de posibles secuencias finitas de instrucciones, que uno puede escribir, (digamos que usando un teclado estándar) es numerable. Pero existe una cantidad incontable de conjuntos de números naturales (por el argumento diagonal de Cantor). Se sigue que casi todos los conjuntos, en un sentido, son *indecidibles*.

El hecho de que no todo conjunto es decidible es relevante a la teoría de la computación. El hecho de que existe un límite a lo que puede ser realizado por un procedimiento efectivo significa que existe un límite a lo que puede -aún en principio- hacerse mediante programas de computadora. Y esto dispara las preguntas: ¿Qué se puede hacer? ¿Qué no se puede hacer?

Históricamente, la teoría de la computabilidad surgió antes del desarrollo de las computadoras digitales. La teoría de la computabilidad es relevante a ciertas consideraciones en lógica matemática. En el corazón de la actividad matemática está la demostración de teoremas. Considere lo que se requiere para que una cadena de símbolos constituya una "demostración matemática aceptable". Antes de aceptar una demostración, y agregar el resultado que fue demostrado a nuestro almacén de conocimiento matemático, insistimos en que la demostración sea *verificable*.

Esto es, debe ser posible que otro matemático, tal como el árbitro del artículo que contiene la demostración, cheque, paso a paso, la corrección de la demostración. Eventualmente, el árbitro o concluye que la demostración en realidad es correcta o concluye que la prueba contiene un hueco o un error y que por lo tanto aún no es aceptable. Esto es, el conjunto de demostraciones matemáticas aceptables -pensadas como cadenas de símbolos- debe ser decidible. Este hecho veremos (en un capítulo posterior) que tiene consecuencias significativas para lo que puede y no puede ser demostrado. Concluimos que la teoría de la computabilidad es relevante a los fundamentos de las matemáticas. Pero si los lógicos no hubieran inventado el concepto de computabilidad, los computólogos lo hubieran hecho más tarde.

1.1.2. Funciones calculables

Antes de continuar, debemos ampliar nuestras consideraciones sobre conjuntos decidibles e indecidibles y extenderlas a la situación más general de las *funciones parciales*. Sea $\mathbb{N} = \{0, 1, 2, \ldots\}$ el conjunto de números naturales. Entonces, un ejemplo de una función de aridad-dos sobre $\mathbb{N}$ es la función sustracción

$$g(m,n) = \begin{cases} m-n & \text{si } m \geqslant n \\ 0 & \text{en otro caso} \end{cases}$$

(donde hemos evitado números negativos). Una función sustracción diferente es la función "parcial"

$$f(m,n) = \begin{cases} m-n & \text{si } m \geqslant n \\ \uparrow & \text{en otro caso} \end{cases}$$

donde "$\uparrow$" indica que la función está indefinida. Así $f(5,2) = 3$, pero $f(2,5)$ está indefinida, el par $(2,5)$ no está en el dominio de f.

En general, decimos que una *función parcial* de aridad-k sobre $\mathbb{N}$ es una función cuyo dominio es algún conjunto de tuplas-k de números naturales y cuyos valores son números naturales. En otras palabras, para una función parcial f de aridad-k y una tupla-k $\langle x_1, \ldots, x_k \rangle$, posiblemente $f(x_1, \ldots, x_k)$ esté definida (es decir, $\langle x_1, \ldots, x_k \rangle$ está en el dominio de f), en cuyo caso el valor de la función está en $\mathbb{N}$, y posiblemente $f(x_1, \ldots, x_k)$ esté indefinida (es decir, $\langle x_1, \ldots, x_k \rangle$ no está en el dominio de f).

En un extremo, existen funciones parciales cuyos dominios son el conjunto $\mathbb{N}^k$ de todas las tuplas-k; de tales funciones se dice que son *totales*. (El adjetivo "parcial" cubre tanto a las funciones que son totales como a las funciones que no son totales.) En el otro extremo, existe la función vacía, esto es, la función que está definida en ninguna parte. La función vacía puede no parecer particularmente útil, pero cuenta como una de las funciones parciales de aridad-k.

Para una función parcial f de aridad-k, decimos que f es una *función parcial calculable efectivamente* si existe un procedimiento efectivo con la siguiente propiedad:

- Dada una tupla-k $\vec{x}$ en el dominio de f, el procedimiento eventualmente regresa el valor correcto de $f(\vec{x})$ y para.
- Dada una tupla-k $\vec{x}$ *no* en el dominio de f, el procedimiento no regresa un valor y no para.

(Aquí hay detalle: ¿Cómo se puede dar un número? Para comunicar un número x al procedimiento, enviamos el *numeral* para x). Los numerales son trozos de lenguage que se pueden comunicar. Los números no. La comunicación requiere lenguage. Sin embargo, continuaremos hablando de sean "m y n números dados" y así sucesivamente. Pero en unos pocos puntos, necesitaremos ser más exáctos y tomar en cuenta el hecho de que a un procedimiento se le dan numerales. Hubo un tiempo en los 1960s cuando, como parte de la "nueva matemática", a los profesores de educación básica se les animó para que distinguieran cuidadosamente entre números y numerales. Esta fue una buena idea que no funcionó.)

Por ejemplo, la función parcial para sustracción

$$f(m,n) = \begin{cases} m-n & \text{si } m \geqslant n \\ \uparrow & \text{en otro caso} \end{cases}$$

es calculable efectivamente, los procedimientos para calcularla, usando numerales en base-10, se enseñan en educación básica.

La función vacía es calculable efectivamente. El procedimiento efectivo para ella, dada una tupla-k, no necesita hacer algo en particular. Pero no debe parar ni regresar un valor.

El concepto de decidibilidad puede ahora ser descrito en términos de funciones. Para un subconjunto S de $\mathbb{N}^k$, podemos decir que S es *decidible* ssi su función característica

$$C_S(\vec{x}) = \begin{cases} \text{Sí} & \text{si } \vec{x} \in S \\ \text{No} & \text{si } \vec{x} \notin S \end{cases}$$

(la cual siempre es total) es calculable efectivamente. Aquí "Sí" y "No" son números fijos de $\mathbb{N}$, tales como 1 y 0.

(Esa palabra "ssi" en el párrafo anterior significa "si y sólo si". Esta es un trozo de jerga matemática que ha demostrado ser tan útil que se ha convertido en una parte estándar de la forma de hablar en matemáticas.)

Aquí, si $k = 1$, entonces S es un conjunto de números. Si $k = 2$, entonces tenemos el concepto de una relación binaria decidible sobre números, y así sucesivamente. Tome, por ejemplo, la relación de divisibilidad, esto es, el conjunto de pares $\langle m, n \rangle$ tales que m divide a n exactamente. (Para que todo esté definido, asuma que 0 divide sólo a sí mismo.) La relación de divisibilidad es decidible ya que, dados m y n, podemos efectuar el algoritmo de división que todos aprendimos en cuarto año y ver si el residuo es cero o distinto de cero.

Ejemplo 1.1.1 Cualquier función total constante sobre $\mathbb{N}$ es calculable efectivamente. Suponga, for ejemplo, $f(x) = 36$ para todo $x \in \mathbb{N}$. Hay un procedimiento obvio para calcular f; ignora la entrada y escribe "36" como salida. Esto puede parecer una trivialidad, pero compárelo con el siguiente ejemplo.

Ejemplo 1.1.2 Defina la función F como sigue

$$F(x) = \begin{cases} 1 & \text{si la conjetura de Goldbach es verdadera} \\ 0 & \text{si la conjetura de Goldbach es falsa} \end{cases}$$

La conjetura de Goldbach enuncia que todo entero par mayor que 2 es la suma de dos primos; por ejemplo, $22 = 5 + 17$. Esta conjetura sigue

siendo un problema abierto en matemáticas. ¿Esta función F es calculable efectivamente? (Elija su respuesta antes de leer el siguiente párrafo.)

Observe que F es una función total constante. (La lógica clásica entra aquí: O existe un número par que sirva como contraejemplo o no existe.) Así como se señaló en el ejemplo anterior, F es calculable efectivamente. ¿Cuál es entonces un procedimiento para calcular F? No lo sé, pero te puedo dar dos procedimientos teniendo la confianza de que uno de ellos calcula F.

El punto de este ejemplo es que la calculabilidad efectiva es una propiedad de la función en sí misma, no es una propiedad de alguna descripción lingüística usada para especificar la función. (Uno dice que la propiedad de calculabilidad efectiva es *extensional*.) Existen muchas frases en Español que servirían para definir F. Para que una función sea calculable efectivamente, debe existir (en sentido matemático) un procedimiento efectivo para calcularla. Que no es lo mismo que decir que tienes ese procedimiento en la mano. Si en el año 2083, alguna criatura en el universo prueba (o refuta) la conjetura de Goldbach, entonces eso no significa que F cambie de repente de no calculable a calculable. Siempre fue calculable.

No obstante, posteriormente habrá situaciones en las que querremos más que la mera existencia un procedimiento efectivo P; querremos alguna manera para encontrar realmente a P, dadas algunas pistas apropiadas. Esto es para después.

Es muy natural extender estos conceptos a la situación donde tenemos la mitad de decidibilidad: Decimos que S es *semidecidible* si su "función semicaracterística"

$$c_S(\vec{x}) = \begin{cases} \text{Sí} & \text{si } \vec{x} \in S \\ \uparrow & \text{si } \vec{x} \notin S \end{cases}$$

es una función parcial calculable efectivamente. Así, un conjunto S de números es semidecidible si existe un procedimiento efectivo para *reconocer* a los miembros de S. Podemos pensar en S como el conjunto que el procedimiento *acepta*. Y el procedimiento efectivo, si bien no es un procedimiento de decisión, es al menos un procedimiento de *aceptación*.

Cualquier conjunto decidible es también semidecidible. Si tenemos un procedimiento efectivo que calcula la función característica C_S, entonces podemos convertirlo en un procedimiento efectivo que calcule la función semicaracterística c_S. Simplemente reemplazamos cada instrucción "escribe No" por un ciclo interminable. O más informalmente, simplemente destornillamos el foco No.

¿Qué sobre la inversa? ¿Existen conjuntos semidecidibles que no son decidibles? Veremos que en verdad existen. El problema con la función semicaracterística es que nunca produce una respuesta No. Suponga que hemos estado calculando $c_S(\vec{x})$ por 37 años y el procedimiento aún no ha terminado. ¿Debemos rendirnos y concluir que $\vec{x}$ no está en S? O quizás trabajando sólo otros diez minutos produciría la información de que $\vec{x}$ pertenece a S. No hay, en general, manera de saberlo.

Aquí hay otro ejemplo de una función parcial calculable:

$$F(n) = \text{el } p > n \text{ más pequeño tal que tanto } p \text{ como } p+2 \text{ son primos}$$

Aquí se ha de entender que $F(n)$ está indefinido si no existe un número p como se describió; así F podría no ser total. Por ejemplo, $F(9) = 11$ ya que ambos 11 y 13 son primos. No se sabe si F es total o no. La "conjetura de los primos gemelos", la que dice que existen infinitos pares de primos que difieren en 2, es equivalente a la afirmación de que F es total. La conjetura de los primos gemelos es aún un problema abierto. Sin embargo, podemos estar seguros de que F es calculable efectivamente. Un procedimiento para calcular $F(n)$ es como sigue. "Dado n, primero defina $p = n + 1$. Luego verifique si p y $p + 2$ son o no primos. Si lo son, entonces pare y dé la salida p. Si no, incremente p y continue." ¿Qué si n es enorme, digamos, $n = 10^{10^{10}}$? Por un lado si existe un par de primos más grande, entonces este procedimiento encontrará al primero y para con la salida correcta. Por otro lado, si no hay un par de primos más grande, entonces el procedimiento nunca para, así nunca da una respuesta. Lo que es correcto; ya que $F(n)$ está indefinida, el procedimiento no debe darnos alguna respuesta.

Ahora suponga que modificamos el ejemplo. Considere la función total:

$$G(n) = \begin{cases} F(n) & \text{si } F(n)\downarrow \\ 0 & \text{en otro caso} \end{cases}$$

Aquí "$F(n)\downarrow$" significa que $F(n)$ está definida, así que n pertenece al dominio de F. Entonces la función G *también* es calculable efectivamente. Esto es, *existe* un programa que calcula G correctamente.

La conjetura de los primos gemelos es verdadera o falsa: O hay infinitos pares de primos, o hay un par más grande. (En este punto, la lógica clásica entra una vez más). En el primer caso, $F = G$ y el procedimiento efectivo para F también calcula G. En el segundo caso, G es eventualmente la función

constante 0. Y cualquier función eventualmente constante es calculable (el procedimiento puede utilizar una tabla para la parte finita de la función antes de que se estabilice).

Así en cualquier caso, *existe* un procedimiento efectivo para G. Eso no es lo mismo que conocer el procedimiento. Este ejemplo indica una vez más la diferencia entre conocer que un cierto procedimiento efectivo existe y tener el procedimiento efectivo en nuestras manos (o tener razones convincentes para saber que el procedimiento en nuestras manos funcionará).

Un programa de una persona es otro dato de esa persona. Éste es el principio detrás de los sistemas operativos (y detrás de la idea de un programa de computadora almacenado). El programa favorito de uno es, para el sistema operativo, otra pieza de dato para ser recibida como entrada y procesarla. El sistema operativo está calculando los valores de una función "universal" de aridad-2. (Históricamente, ¡el flujo de ideas fue exactamente en la dirección opuesta! La siguiente digresión expande este punto).

Digresión: El concepto de programa de computadora de propósito general almacenado es ahora muy común, pero el concepto se desarrolló lentamente sobre un periodo de tiempo. ¡La máquina ENIAC, la computadora más importante en los 1940s, era programada moviendo interruptores e insertando cables en tableros con contactos! Esto está muy lejos de tratar un programa como dato. Fue von Neumann quien, en un reporte técnico de 1945, estableció las ideas cruciales para un programa de computadora de propósito general almacenado, esto es, para una computadora universal. El artículo de Turing de 1936 sobre lo que ahora es llamado máquinas de Turing ha demostrado la existencia de una "máquina universal de Turing" para calcular la función Φ descrita posteriormente. Cuando Turing fue a Princeton en 1936-37, von Neumann estubo ahí y debió de estar consciente de su trabajo. Aparentemente, el pensamiento de von Neumann en 1945 fue influenciado por el trabajo de Turing de casi una década antes.

Suponga que adoptamos un método fijo para codificar cualquier conjunto de instrucciones mediante un solo número natural. (Primero, convertimos las instrucciones a una cadena de 0's y 1's -uno siempre hace esto con programas de computadora- y luego consideramos a la cadena como el nombre de un número natural en notación base-2.) Entonces la "función universal"

$$\Phi(w, x) = \begin{array}{l}\text{el resultado de aplicar las instrucciones} \\ \text{codificadas mediante } w \text{ a la entrada } x\end{array}$$

es una función parcial calculable efectivamente (donde se entiende que $\Phi(w, x)$

está indefinida si al aplicar las instrucciones codificadas mediante w a la entrada x no para y no regresa una salida). Aquí están las instrucciones para Φ: "Dados w y x, decodifica w para ver que hay que hacer con x y luego lo hace". Por supuesto, la función Φ no es total. Por una cosa, cuando tratamos de decodificar w, podriamos encontrar un completo sin sentido, así que la instrucción "luego lo hace" nos lleva a ningún lado. Y aún si al decodificar w obtenemos instrucciones explícitas y comprensibles, el aplicar esas instrucciones a un x en particular podría nunca producir una salida.

(Este razonamiento será repetido en el Capítulo 3, cuando tengamos material más concreto con el que tratar. Pero las ideas rectoras serán las mismas.)

La función parcial de aridad-dos Φ es "universal" en el sentido de que *cualquier* función parcial f efectivamente calculable está dada por la ecuación

$$f(x) = \Phi(e, x) \text{ para todo } x$$

donde e codifica las instrucciones de f. Será útil introducir aquí una notación especial: Sea $[\![e]\!]$ la función parcial de aridad-uno definida por la ecuación

$$[\![e]\!](x) = \Phi(e, x)$$

Esto es, $[\![e]\!]$ es la función parcial cuyas instrucciones están codificadas por e, en el entendido de que algunos valores de e podrían codificar algo no sensato, la función $[\![e]\!]$ podría ser la función vacía. En cualquier caso, $[\![e]\!]$ es la función parcial que obtenemos de Φ, cuando fijamos su primer variable en e. Así,

$$[\![0]\!], [\![1]\!], [\![2]\!], \ldots$$

es una lista completa (con repeticiones) de todas las funciones parciales calculables efectivamente. Los valores de $[\![e]\!]$ están dados por el $(e+1)-$ésimo renglón en la siguiente tabla:

$[\![0]\!]$	$\Phi(0,0)$	$\Phi(0,1)$	$\Phi(0,2)$	$\Phi(0,3)$	$\cdots$
$[\![1]\!]$	$\Phi(1,0)$	$\Phi(1,1)$	$\Phi(1,2)$	$\Phi(1,3)$	$\cdots$
$[\![2]\!]$	$\Phi(2,0)$	$\Phi(2,1)$	$\Phi(2,2)$	$\Phi(2,3)$	$\cdots$
$[\![3]\!]$	$\Phi(3,0)$	$\Phi(3,1)$	$\Phi(3,2)$	$\Phi(3,3)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Usando la función parcial universal Φ, podemos construir una relación binaria *indecidible*, la relación *de paro* H:

$$\begin{aligned} \langle w, x\rangle \in H &\Leftrightarrow \Phi(w, x)\downarrow \\ &\Leftrightarrow \text{al aplicar las instrucciones codificadas por } w \\ &\quad\ \text{a la entrada } x \text{ para} \end{aligned}$$

Desde el lado positivo, H es semidecible. Para calcular la función semicaracterística $c_H(w, x)$, dados w y x, primero calculamos $\Phi(w, x)$. Si cuando ésta regrese un valor y pare, damos la salida "Sí" y paramos.

Desde el lado negativo, H no es decidible. Para ver esto, primero considere la siguiente función parcial:

$$f(x) = \begin{cases} \text{Sí} & \text{si } \Phi(x, x)\uparrow \\ \uparrow & \text{si } \Phi(x, x)\downarrow \end{cases}$$

(Note que estamos usando la construcción diagonal clásica. Mirando la anterior tabla de los valores de Φ ordenados en un arreglo bi-dimensional, uno ve que f ha sido construida yendo a lo largo de la diagonal de la tabla, tomando la entrada $\Phi(x, x)$ encontrada allí, y asegurando que $f(x)$ difiera de ella).

Hay dos cosas que decir sobre f. Primero, f no puede ser calculable efectivamente. Considere cualquier conjunto de instrucciones que pudiera calcular f. Dichas instrucciones tienen algún número de código k y así calcula la función $[\![k]\!]$. ¿Pudiera ser la misma que f? No, f y $[\![k]\!]$ difieren en la entrada k. Esto es, f ha sido construida de tal manera que $f(k)$ difiere de $[\![k]\!](k)$, ellas difieren ya que una está definida y la otra no. Así estas instrucciones no pueden calcular correctamente a f; ellas producen el resultado equivocado en la entrada k. Y como k fue arbitrario, estamos forzados a concluir que *ningún* conjunto de instrucciones puede calcular correctamente a f. (Este es nuestro primer ejemplo de una función parcial que no es calculable efectivamente. Existen muchísimas más, como veremos).

En segundo lugar, podemos argumentar que *si* tuvieramos un procedimiento de decisión para H, entonces podriamos calcular f. Para calcular $f(x)$, usamos primero el procedimiento de decisión H para decidir si $\langle x, x\rangle \in H$ o no. Si no, entonces $f(x) =$ Sí. Pero si $\langle x, x\rangle \in H$, entonces el procedimiento para encontrar $f(x)$ caería en un ciclo infinito ya que $f(x)$ está indefinida.

Juntando estas dos observaciones sobre f, concluimos que no puede existir un procedimiento de decisión para H. El hecho de que H es indecidible

se expresa usualmente diciendo que "el problema de paro es irresoluble o también que es insoluble"; es decir, no podemos en general determinar efectivamente, dados w y x, si al aplicar las instrucciones codificadas por w a la entrada x eventualmente terminará o seguirá por siempre: **Insolubilidad del problema de paro:** La relación

$$\{\langle w, x\rangle | \text{ al aplicar las instrucciones codificadas por } w \text{ a la entrada } x \text{ para}\}$$

es semidecidible pero no decidible.

La función f del argumento anterior

$$f(x) = \begin{cases} \text{Sí} & \text{si } \Phi(x,x) \uparrow \\ \uparrow & \text{si } \Phi(x,x) \downarrow \end{cases}$$

es la función semicaracterística del conjunto $\{x|\Phi(x,x) \uparrow\}$. Ya que su función semicaraterística no es calculable efectivamente, podemos concluir que este conjunto *no* es semidecidible.

Sea K el complemento de este conjunto:

$$K = \{x|\Phi(x,x) \downarrow\} = \{x|[\![x]\!](x) \downarrow\}$$

Este conjunto es semidecidible. ¿Cómo podriamos calcular $c_K(x)$, dado x? Tratamos de calcular $\Phi(x,x)$ (lo cual es posible ya que Φ es una función parcial calculable efectivamente). Si y cuando el cálculo regrese una salida y pare, damos la salida "Sí" y paramos. Hasta ese momento seguimos intentando. (Este argumento es el mismo que vimos para la semidecidibilidad de H. Y $x \in K \Leftrightarrow \langle x, x\rangle \in H$).

Teorema de Kleene: *Un conjunto (o una relación) es decidible si y sólo si él y su complemento son semidecidibles.*

Si estamos trabajando con conjuntos de números, entonces el complemento es con respecto a $\mathbb{N}$; si estamos trabajando con una relación de aridad-k, entonces el complemento es con respecto a $\mathbb{N}^k$.

Prueba: Por un lado, si un conjunto S es decidible, entonces su complemento $\overline{S}$ también es decidible, simplemente intercambiamos el "Sí" y el "No". Así tanto S como su complemento $\overline{S}$ son semidecidibles ya que los conjuntos decidibles son también semidecidibles.

Por otro lado, suponga que S es un conjunto para el que tanto c_S como $c_{\overline{S}}$ son calculables efectivamente. La idea es poner juntas estas dos mitades de un procedimiento de decisión para hacer uno completo. Digamos que queremos

encontrar $C_S(x)$, dado x. Necesitamos organizar nuestro tiempo. Durante los minutos impares, corremos nuestro programa para $c_S(x)$. Durante los minutos pares, corremos nuestro programa para $c_{\overline{S}}(x)$. Por supuesto, al final de cada minuto, almacenamos por ahí lo que hemos hecho, así podemos posteriormente recogerlo de donde lo hayamos dejado.

Eventualmente, debemos recibir un "Sí". Si es durante un minuto impar, encontramos que $c_S(x) =$ Sí (esto debe suceder eventualmente si $x \in S$), luego damos la salida "Sí" y paramos. Y si es durante un minuto par, encontramos que $c_{\overline{S}}(x) =$ Sí (esto debe suceder eventualmente si $x \notin S$), luego damos la salida "No" y paramos.

(Alternativamente, podemos imaginar que trabajamos de manera ambidiestra. Con la mano izquierda, trabajamos calculando $c_S(x)$; con la mano derecha, trabajamos con $c_{\overline{S}}(x)$. Eventualmente, una mano descubre la respuesta). $\dashv$

El conjunto K es un ejemplo de un conjunto semidecidible que no es decidible. Su complemento $\overline{K}$ no es semidecidible; hemos visto que su función semicaracterística f no es calculable efectivamente.

La conexión entre funciones parciales calculables efectivamente y conjuntos semidecidibles puede describirse mejor como sigue:
Teorema:

(i) *Una relación es semidecidible si y sólo si es el dominio de alguna función parcial calculable efectivamente.*

(ii) *Una función parcial f es una función parcial calculable efectivamente si y sólo si su gráfica G (es decir, el conjunto de tuplas $\langle \vec{x}, y \rangle$ tales que $f(\vec{x}) = y$) es una relación semidecidible.*

Prueba: Para la proposición (i), en una dirección es verdadera por definición: Cualquier relación es el dominio de su función característica, y para una relación semidecidible, esa función es una función parcial calculable efectivamente.

Recíprocamente, para una función parcial calculable efectivamente, f, tenemos el procedimiento de semidecisión natural para su dominio: Dado $\vec{x}$, tratamos de calcular $f(\vec{x})$. Si y cuando tengamos éxito para encontrar $f(\vec{x})$, ignoramos el valor y simplemente decimos Yes y paramos.

Para probar (ii) en una dirección, suponga que f es una función parcial calculable efectivamente. Aquí tenemos un procedimiento de semidecisión para su gráfica G: Dado $\langle \vec{x}, y \rangle$, procedemos a calcular $f(\vec{x})$. Si y cuando

obtenemos el resultado, chacamos para ver si es y o no. Si el resultado en realidad es y, entonces decimos Sí y paramos.

Por supuesto, este procedimiento falla al tratar de dar una respuesta si $f(x) \uparrow$, lo cual es exactamente como debe ser, ya que en este caso, $\langle \vec{x}, y \rangle$ no está en la gráfica.

Para probar la otra dirección de (ii) , suponga que tenemos un procedimiento de semidecisión para la gráfica G. Buscamos calcular, dado $\vec{x}$, el valor $f(\vec{x})$, si éste está definido. Nuestro plan es checar $\langle \vec{x}, 0 \rangle, \langle \vec{x}, 1 \rangle, \langle \vec{x}, 2 \rangle, \ldots$, para ver su membresia a G. Pero para organizar nuestro tiempo sensatamente, usamos un procedimiento llamado "machihembrar". Aquí está lo que hacemos:

1. Usa un minuto probando si $\langle \vec{x}, 0 \rangle \in G$.

2. Usa dos minutos probando si $\langle \vec{x}, 0 \rangle \in G$ y dos minutos probando si $\langle \vec{x}, 1 \rangle \in G$.

3. Similarmente, usa tres minutos en cada uno de $\langle \vec{x}, 0 \rangle, \langle \vec{x}, 1 \rangle$ y $\langle \vec{x}, 2 \rangle$.

Y así sucesivamente. Si y cuando descubramos que, en efecto, $\langle \vec{x}, k \rangle \in G$, entonces regresamos el valor k y paramos. Observe que siempre que $f(\vec{x}) \downarrow$, tarde o temprano el procedimiento anterior determinará correctamente $f(\vec{x})$ y parará. Por supuesto, si $f(x) \uparrow$, entonces el procedimiento se ejecutará por siempre. $\dashv$

1.1.3. Tesis de Church

No obstante el concepto de calculabilidad efectiva ha sido descrito aquí en términos algo vagos, en la siguiente sección describiremos un concepto preciso (matemáticamente) de una "función parcial computable". En efecto, se describirán varias maneras equivalentes para formular el concepto en términos precisos. Se argumentará que el concepto matemático de función parcial computable es la formalización *correcta* del concepto informal de una función parcial calculable efectivamente. Esta afirmación es conocida como la *tesis de Church* o la *tesis de Church-Turing*.

La tesis de Church, relaciona una idea informal con una idea formal, no es en sí una proposición matemática capaz de ser probada. Pero uno puede buscar evidencia a favor o en contra de la tesis de Church, todo resulta ser evidencia a favor.

Una pieza de evidencia es la ausencia de contraejemplos. Esto es, cualquier función examinada hasta ahora que los matemáticos han creido que es calculable efectivamente, se ha encontrado que es computable.

Evidencia más fuerte deriva de varios intentos que diferentes personas hicieron independientemente, tratando de formalizar la idea de calculabilidad efectiva. Alonzo Church uso el cálculo$-\lambda$; Alan Turing uso un agente de cálculo idealizado (posteriormente llamado maquina de Turing); Emil Post desarrollo un enfoque similar. Extraordinariamente, todos estos intentos resultaron ser equivalentes, en que todos ellos definieron exactamente la misma clase de funciones, llamadas ¡funciones parciales computables! . El estudio de la calculabilidad efectiva se inició en 1930s con el trabajo en lógica matemática. Como se ha notado previamente, el asunto está relacionado con el concepto de una *prueba aceptable*. Más recientemente, el estudio de la calculabilidad efectiva ha formado una parte escencial de la teoría de la computación. Un científico de la computación prudente seguramente querrá saber que, además de las dificultades que el mundo real presenta, existe un límite puramente teórico para la calculabilidad.

1.1.4. Ejercicios

Ejercicios 1.1.1 Realice lo que se indica:

1. Asuma que S es un conjunto de número naturales pero que es finito. (Esto es, S es un subconjunto cofinito de $\mathbb{N}$). Explique por qué S debe ser decidible.

2. Asuma que A y B son conjuntos decidibles de números naturales. Explique por qué su intersección $A \cap B$ es también decidible. (Describa un procedimiento efectivo para determinar si un número dado está o no en $A \cap B$.)

3. Asuma que A y B son conjuntos decidibles de números naturales. Explique por qué su unión $A \cup B$ es también decidible.

4. Asuma que A y B son conjuntos semidecidibles de números naturales. Explique por qué su intersección $A \cap B$ es también semidecidible.

5. Asuma que A y B son conjuntos semidecidibles de números naturales. Explique por qué su unión $A \cup B$ es también semidecidible.

6. *a*) Asuma que R es una relación binaria decidible sobre los números naturales. Esto es, es una relación decidible de aridad-2. Explique por qué su dominio, $\{x|\langle x, y\rangle \in R \text{ para algún } y\}$, es un conjunto semidecidible.

   b) Ahora suponga que en vez de asumir que R es decidible, asumimos sólo que es semidecidible. ¿Sigue siendo cierto que su dominio debe ser semidecidible?

7. *a*) Asuma que f es una función total calculable de aridad-uno. Explique por qué su gráfica es una relación binaria decidible.

   b) Inversamente, muestre que si la gráfica de una función total de aridad-uno f es decidible, entonces f debe ser calculable.

   c) Ahora asuma que f es una función parcial calculable de aridad-uno, no necesariamente total. Explique por qué su dominio, $\{x \in \mathbb{N}|f(x)\downarrow\}$, es semidecidible.

8. Asuma que S es un conjunto decidible de números naturales y que f es una función *total* calculable efectivamente sobre $\mathbb{N}$. Explique por qué $\{x|f(x) \in S\}$ es decidible. (Este conjunto es llamado la *imagen inversa* de S bajo f.)

9. Asuma que S es un conjunto semidecidible de números naturales y que f es una función parcial calculable efectivamente sobre $\mathbb{N}$. Explique por qué
$$\{x|f(x)\downarrow \text{ y } f(x) \in S\}$$
es semidecidible.

10. En la expansión decimal de π, podría haber una cadena de algunos 7's consecutivos. Defina la función f tal que $f(x) = 1$ si existe una cadena de x o más 7's y $f(x) = 0$ en otro caso:
$$f(x) = \begin{cases} 1 & \text{si } \pi \text{ tiene una cadena con } x \text{ o más 7's} \\ 0 & \text{en otro caso} \end{cases}$$
Explique, sin usar algún hecho especial sobre π o teoría de números, por qué f es calculable efectivamente.

11. Asuma que g es una función total no creciente (esto es, $g(x) \geqslant g(x+1)$ para todo x) sobre $\mathbb{N}$. Explique por qué g debe ser calculable efectivamente.

12. Asuma que f es una función total sobre los números naturales y que f es eventualmente periódica. Esto es, existen números m y p tales que para todo x mayor que m, tenemos que $f(x+p) = f(x)$. Explique por qué f es calculable efectivamente.

13. *a*) Asuma que f es una función total calculable efectivamente sobre los números naturales. Explique por qué el rango de f (esto es, el conjunto $\{f(x)|x \in \mathbb{N}\}$) es semidecidible.

    b) Ahora suponga que f es una función parcial calculable efectivamente (no necesariamente total). ¿Sigue siendo cierto que su rango debe ser semidecidible?

    c) Asuma que f y g son funciones parciales calculables efectivamente sobre $\mathbb{N}$. Explique por qué el conjunto

    $$\{x|f(x) = g(x) \text{ y ambos están definidos}\}$$

    es semidecidible.

1.2. Formalizaciones - Un Panorama

En la sección anterior, el concepto de calculabilidad efectiva se describió muy informalmente. Ahora queremos hacer aquellas ideas precisas (es decir, hacerlas parte de las matemáticas). En efecto, varios enfoques para hacerlo serán descritos: dispositivos de cálculo idealizados, definiciones generativas (es decir, la clase más pequeña que contiene ciertas funciones iniciales y que es cerrada bajo ciertas construcciones), lenguajes de programación y definibilidad en lenguajes formales. Es un hecho significativo que estos enfoques, a pesar de ser muy distintos, producen conceptos exactamente equivalentes.

Esta sección da un panorama general de un número de maneras diferentes (pero equivalentes) de formalizar los conceptos de calculabilidad efectiva. En capítulos posteriores se desarrollarán con todo detalle algunas de estas maneras.

Digresión: El libro de Rogers de 1967 citado en las Referencias demostró que el concepto de computabilidad se puede desarrollar *sin* adoptar alguna

de estas formalizaciones. Ese libro fue precedido por una versión preliminar mimeografiada en 1956, en la que primero vi este concepto. Todavía existen pocas copias atesoradas de la edición mimeografiada.

1.2.1. Máquinas de Turing

A principios de 1935, Alan Turing era un estudiante de 22 años de edad graduado en el King's College en Cambridge. Bajo la asesoría de Max Newman, estuvo trabajando en el problema de formalizar el concepto de calculabilidad efectiva. En 1936, se enteró del trabajo de Alonzo Church, en Princeton. Church también había estado trabajando sobre el mismo problema, y en su artículo de 1936, "Un problema insoluble de teoría elemental de números", presentó la afirmación categórica de que la clase de funciones calculables efectivamente debe ser idéntica a la clase de funciones definibles en el *cálculo lambda*, un lenguaje formal para especificar la construcción de funciones. Aún más, Church mostró que exactamente la misma clase de funciones puede ser caracterizada en términos de derivabilidad formal de ecuaciones.

Luego Turing terminó de escribir rápidamente su artículo, en el cual presentó un enfoque muy diferente para caracterizar las funciones calculables efectivamente, pero uno que -como demostró- una vez más producía la misma clase de funciones que Church había propuesto. Con el estímulo de Newman, Turing fue a Princeton por dos años, donde escribió su tesis doctoral bajo la dirección de Alonzo Church.

El artículo de Turing sigue siendo una introducción a sus ideas muy legible. ¿Cómo podría un empleado diligente realizar un cálculo siguiendo instrucciones? Él o ella podría organizar el trabajo en una libreta. En cualquier momento dado, su atención se centra en una página en particular. Siguiendo sus instrucciones, podría alterar la página, y entonces podría cambiar a otra página. Y como la libreta es suficientemente grande (o el suministro de papel nuevo es suficientemente basto) nunca llega a la última página.

El alfabeto de símbolos disponibles para el empleado debe ser finito; si hubieran infinitos símbolos, entonces habrían dos que serían arbitrariamente similares y se podría confundir. Luego podemos sin perdida de generalidad pensar que podemos escribir sobre una página de la libreta un solo símbolo. Y podemos imaginar que las páginas de la libreta están puestas una al lado de la otra, formando una cinta de papel, que consiste de cuadrados, cada cuadrado está en blanco o tiene impreso un símbolo. (Por uniformidad, podemos pensar que un cuadrado blanco contiene el símbolo "blanco" B). En cada etapa de su

trabajo, el empleado -o la máquina mecánica- puede alterar el cuadrado que está examinando, puede poner atención al siguiente cuadrado o al previo, y puede ver las instrucciones para saber cual de ellas debe seguir a continuación. Turing describió la última parte como un "cambio de estado mental".

Turing escribió, "Ahora podemos construir una máquina para hacer el trabajo". Tal máquina es, por supuesto, ahora llamada *máquina de Turing*, una frase que usó primero Church en su revisión al artículo de Turing que apareció en *The Journal of Simbolic Logic*. La máquina tiene una cinta potencialmente infinita. Inicialmente el numeral o palabra dada de entrada se escribe en la cinta, la cual es blanca en los restantes cuadrados. La máquina es capaz de estar en cualquiera de los "estados" finitos (la palabra "mental" es inapropiada para un máquina).

En cada paso del cálculo, dependiendo del estado en que se encuentre en el momento, la máquina puede cambiar el símbolo en el cuadrado que está examinando en ese momento, y puede dirigir su atención al cuadrado a la izquierda o la derecha, y puede entonces cambiar su estado a otro estado. (La cinta se extiende sin fin en ambas direcciones).

El programa para esta máquina de Turing se puede dar mediante una tabla. Donde los posibles estados de la máquina son $q_1, \ldots, q_r$, cada línea de la tabla es una quíntupla $\langle q_i, S_j, S_k, D, q_m \rangle$ la que será interpretada como que siempre que la máquina esté en el estado q_i y el cuadrado examinado contiene el símbolo S_j, entonces el símbolo debe ser cambiado a S_k y la máquina debe cambiar su atención al cuadrado a la izquierda (si $D = L$) o a la derecha (si $D = R$), y debe cambiar su estado a q_m. Posiblemente el símbolo S_j es el símbolo "blanco" B, lo que significa que el cuadrado examinado es blanco; posiblemente S_k es B, lo que significa que lo que sea que esté en el cuadrado tiene que ser borrado.

Para que el programa no sea ambiguo, no deben haber dos quintuplas diferentes con los primeros dos componentes iguales. (Al relajar el requerimiento con respecto a la ausencia de ambigüedad, obtenemos el concepto de máquina de Turing *no determinista*, la que será útil posteriormente, en la discusión de computabilidad factible). Uno de los estados, digamos, q_1, es designado como el estado inicial, el estado en el que la máquina empieza a calcular. Si empezamos a ejecutar la máquina en este estado, y examinamos el primer cuadrado de su entrada, podría (o tal vez no), después de algún número de pasos, alcanzar un estado y un símbolo para los cuales su tabla carece de una quintupla que tenga ese estado y ese símbolo como sus primeros dos componentes. En ese punto, la máquina *para*, y podemos mirar en la

cinta (iniciando con el cuadrado que estaba bajo consideración) para ver que numeral o palabra de salida tiene.

Ahora suponga que Σ es un alfabeto finito (el símbolo B no cuenta como un miembro de Σ). Sea Σ^* el conjunto de todas las palabras sobre este alfabeto (esto es, Σ^* es el conjunto de todas las cadenas, incluyendo a la cadena vacía, que consisten de miembros de Σ). Suponga que f es una función parcial de aridad-k de Σ^* en Σ^*. Diremos que f es *computable a la Turing* si existe una máquina de Turing $\mathcal{M}$ que, cuando empieza en su estado inicial escaneando el primer símbolo de una tupla$-k$ $\vec{w}$ de palabras (escritas sobre la cinta, con un cuadrado blanco entre las palabras, y con el resto de la cinta en blanco), se comporta como sigue:

- Si $f(\vec{w}) \downarrow$ (es decir, si $\vec{w} \in \mathrm{dom} f$) entonces $\mathcal{M}$ eventualmente para, y en ese momento, está escaneando el símbolo más izquierdo de la palabra $f(\vec{w})$ (la cual es seguida por un cuadrado blanco).
- Si $f(\vec{w}) \uparrow$ (es decir, si $\vec{w} \notin \mathrm{dom} f$) entonces $\mathcal{M}$ nunca para.

Ejemplo 1.2.1 Tome un alfabeto de dos letras $\Sigma = \{a, b\}$. Sea $\mathcal{M}$ la máquina de Turing dada por el siguiente conjunto con seis quintuplas[1], donde q_1 es designado como el símbolo inicial:

$$\langle q_1, a, a, R, q_1 \rangle$$
$$\langle q_1, b, b, R, q_1 \rangle$$
$$\langle q_1, B, a, L, q_2 \rangle$$
$$\langle q_2, a, a, L, q_2 \rangle$$
$$\langle q_2, b, b, L, q_2 \rangle$$
$$\langle q_2, B, B, R, q_3 \rangle$$

Suponga que empezamos esta máquina en el estado q_1, escaneando la primer letra de una palabra w. La máquina se mueve (en el estado q_1) hacia el final derecho de w, donde añade la letra a. Entonces se mueve (en el estado q_2) de regreso hacia el final izquierdo de la palabra, donde para (en el estado q_3). Así, $\mathcal{M}$ calcula la función total $f(w) = wa$.

Necesitamos adoptar convenciones especiales para manejar la palabra vacía λ, la cual ocupa cero cuadrados. Esto se puede hacer de diferentes

[1]En el original hay un error, la quinta quintupla aparece como $\langle q_2, b, b, R, q_2 \rangle$. Diga cuál es el comportamiento de esa máquina y qué calcula. N. del T.

maneras, la siguiente es la manera escogida, si la máquina para escaneando un cuadrado blanco, entonces la palabra de salida es λ. Para una función de aridad-uno f, para calcular $f(\lambda)$, simplemente iniciamos con una cinta en blanco.

Para una función de aridad-dos g, para calcular $g(w, \lambda)$, iniciamos sólo con la palabra w, escaneando el primer símbolo de w. Y para calcular $g(\lambda, w)$, también empezamos con sólo la palabra w sobre la cinta, pero escaneando el cuadrado blanco justo a la izquierda de w. Y en general, para darle a una función de aridad-k la entrada $\vec{w} = \langle u_1, \ldots, u_k \rangle$ consistiendo de k palabras de longitudes $n_1, \ldots, n_k$,iniciamos la máquina escaneando el primer cuadrado de la configuración de entrada de longitud $n_1 + \cdots + n_k + k - 1$

$$(n_1 \text{ símbolos de } u_1)B(n_2 \text{ símbolos de } u_2)B \cdots B(n_k \text{ símbolos de } u_k)$$

con el resto de la cinta en blanco. Aquí cualquier n_i puede ser cero; en el caso extremo, todos pueden ser cero.

Un inconveniente obvio de estas convenciones es que no existe diferencia entre el par $\langle u, v \rangle$ y el triple $\langle u, v, \lambda \rangle$. Otras convenciones evitan este inconveniente, a costa de introducir su propia idiosincracia.

La definición de computabilidad a la Turing puede ser adaptada fácilmente a funciones parciales de aridad-k sobre $\mathbb{N}$. La manera más simple de hacerlo es usando numerales en base-1. Tomemos el alfabeto de una letra $\Sigma = \{|\}$ cuya única letra es el símbolo $|$. O para ser más convencionales, sea $\Sigma = \{1\}$, usando el símbolo 1 en lugar de $|$. Entonces la configuración inicial para el triple $\langle 3, 0, 4 \rangle$ es

$$111BB1111$$

Entonces la *tesis de Church*, también llamada -particularmente en el contexto de máquinas de Turing- *tesis de Church-Turing*, es la afirmación de que este concepto de computabilidad a la Turing es la formalización correcta del concepto informal de calculabilidad efectiva. Ciertamente la definición refleja la idea de seguir instrucciones predeterminadas, sin limitar la cantidad de tiempo que puede requerir. (El nombre "tesis de Church-Turing" obscurece el hecho de que Church y Turing siguieron caminos muy distintos para llegar a conclusiones equivalentes).

La tesis de Church ha alcanzado aceptación universal. Kurt Gödel, escribiendo en 1964 sobre el concepto de "sistema formal" en lógica, implicando la idea de que el conjunto de deducciones correctas debe ser un conjunto decidible, dijo que "debido al trabajo de A. M. Turing, ahora se puede dar

una definición precisa e incuestionablemente adecuada del concepto general de sistema formal". Y otros concuerdan.

La robustez del concepto de computabilidad a la Turing se evidencia por el hecho de que es insensible a ciertas modificaciones a la definición de una máquina de Turing. Por ejemplo, podemos imponer limitaciones al tamaño del alfabeto, o podemos insistir en que la máquina nunca se mueva a la izquierda de su punto de inicio. Ninguna de éstas afectará a la clase de funciones parciales computables a la Turing.

Turing desarrollo estas ideas antes de la introducción de las computadoras modernas digitales. Después de la segunda guerra mundial, Turing jugó un papel activo en el desarrollo de las primeras computadoras, y en el campo emergente de la inteligencia artificial. (Durante la guerra, trabajó descifrando el código Enigma usado por los Alemanes en el campo de batalla, trabajo militarmente importante, el cual permaneció clasificado hasta después de la muerte de Turing). Uno puede especular si Turing habría formulado sus ideas de manera diferente, si su trabajo lo hubiera hecho después de la introducción de las computadoras digitales.

Digresión: Hay un ejemplo interesante, que lleva el nombre[2] de "el problema del castor ocupado".

Suponga que queremos una máquina de Turing, iniciando con su cinta en blanco, que escriba tantos unos como pueda, y luego pare. Con un número limitado de estados, ¿cuántos 1's podemos obtener?

Para hacer las cosas más precisas, tome máquinas de Turing con el alfabeto $\{1\}$ (así, los únicos símbolos son B y 1). Permitiremos que tales máquinas tengan n estados, más un estado de paro (puede ocurrir como el último miembro de una quintupla, pero no como el primer miembro). Para cada n, sólo existen finitas máquinas de Turing esencialmente distintas. Algunas de ellas, iniciando con la cinta en blanco, podrían no parar. Por ejemplo, la máquina de un-estado

$$\langle q_1, B, 1, R, q_1 \rangle$$

sigue escribiendo por siempre sin parar. Pero entre aquellas que paran, buscamos las que escriben muchos 1's.

Defina $\sigma(n)$ como el número más grande de 1's que pueden escribirse por una máquina de Turing de n estados como se describió aquí antes de que

[2]Este nombre le ha dado muchas dificultades a los traductores.

pare. Por ejemplo, $\sigma(1) = 1$, ya que la máquina de un-estado

$$\langle q_1, B, 1, R, q_H \rangle$$

(el estado de paro q_H no cuenta) escribe un 1, y ninguna de las otras máquinas de Turing de un-estado lo hacen mejor. (No hay muchas otras máquinas de un-estado, y uno puede examinarlas todas en un plazo de tiempo razonable). Acordemos que $\sigma(0) = 0$. Entonces σ es una función total. También es no decreciente ya que no es un impedimento tener un estado extra para trabajar. A pesar del hecho de que $\sigma(n)$ es meramente el miembro más grande de cierto conjunto finito, no existe un algoritmo que nos permita, en general, evaluarlo.

Ejemplo 1.2.2 Aquí hay un candidato de dos-estados:

$$\langle q_1, B, 1, R, q_2 \rangle$$
$$\langle q_1, 1, 1, L, q_2 \rangle$$
$$\langle q_2, B, 1, L, q_1 \rangle$$
$$\langle q_2, 1, 1, R, q_H \rangle$$

Empezando con una cinta en blanco, esta máquina escribe cuatro 1's consecutivos, y luego para (después de seis pasos), leyendo el tercer 1. Está invitado a verificarlo corriendo la máquina. Concluimos que $\sigma(2) \geqslant 4$.

Teorema 1.2.1 Teorema de Rado (1962): La función σ no es computable a la Turing. Aún más, para cualquier función f computable a la Turing, tenemos que $f(x) < \sigma(x)$ para todo x suficientemente grande. Esto es, σ domina eventualmente a cualquier función total computable a la Turing.

Prueba: Asuma que hemos dado alguna función f computable a la Turing. Debemos mostrar que σ eventualmente la domina. Defina (por razones que pueden parecer inicialmente misteriosas) la función g:

$$g(x) = \text{máx}(f(2x), f(2x + 1)) + 1.$$

Entonces g es total y uno puede mostrar que es computable a la Turing. Así existe alguna máquina de Turing $\mathcal{M}$ con, digamos, k estados que la computa, usando el alfabeto $\{1\}$ y notación base-1. Para cada x, sea $\mathcal{N}_x$ la máquina de Turing de $(x + k)$ estados que primero escribe x 1's sobre la cinta, y luego imita a $\mathcal{M}$. (Los x estados nos permiten escribir x 1's en la cinta de manera directa, y luego están los k estados de $\mathcal{M}$).

Entonces, $\mathcal{N}_x$, cuando inicia con una cinta en blanco, escribe $g(x)$ 1's en la cinta y para. Así, $g(x) \leqslant \sigma(x+k)$, por la definición de σ. Así, tenemos

$$f(2x), f(2x+1) < g(x) \leqslant \sigma(x+k),$$

y si $x \geqslant k$, entonces

$$\sigma(x+k) \leqslant \sigma(2x) \leqslant \sigma(2x+1).$$

Poniendo las dos líneas juntas, vemos que $f < \sigma$ de $2k$ en adelante. $\dashv$

Así σ crece más, eventualmente, que cualquier función total computable a la Turing. ¿Qué tan rápido crece? Entre los números más pequeños, $\sigma(2) = 4$. (El ejemplo precedente mostró que $\sigma(2) \geqslant 4$. La otra desigualdad no es enteramente trivial ya que existen miles de máquinas de dos-estados). También se ha mostrado que $\sigma(3) = 6$ y $\sigma(4) = 13$. De aquí en adelante, sólo son conocidas cotas inferiores. En 1984, se encontró que $\sigma(5)$ es al menos 1915. En 1990, se elevó a 4098. Y $\sigma(6) > 3{,}1 \times 10^{10566}$. Y $\sigma(7)$ debe ser astronómico. Estas cotas inferiores se establecen usando codificaciones ingeniosamente complejas para crear pequeñas máquinas de Turing que escriben muchos 1's y luego paran.

Probar además cotas superiores sería difícil. En efecto, uno puede mostrar, bajo algunas suposiciones razonables, que cotas superiores para $\sigma(n)$ son probables sólo para finitos n's.

Si pudieramos solucionar el problema de paro, tendríamos el siguiente método para calcular $\sigma(n)$:

- Lista todas las máquinas de n-estados.
- Descarta aquellas que nunca paran.
- Ejecuta aquellas que paran.
- Selecciona la puntuación más alta.

El segundo paso es el que nos da problemas. (Nueva información sobre la función σ de Rado sigue descubriéndose. Noticias recientes se pueden obtener de la página Web mantenida por Heiner Marxen, `https://turbotm.de/~heiner/BB/index.html`).

1.2.2. Recursividad primitiva y búsqueda

Para una segunda formalización del concepto de calculabilidad, definiremos cierta clase de funciones parciales sobre $\mathbb{N}$ como la clase más pequeña que contiene ciertas funciones iniciales y es cerrada bajo ciertas construcciones.

Para las funciones iniciales, tomamos las siguientes funciones totales muy simples:

- Las funciones *cero*, esto es, las funciones constantes f definidas por la ecuación:
$$f(x_1, \ldots, x_k) = 0.$$
existe una de tales funciones para cada k.

- La función *sucesor* S, definida por la ecuación:
$$S(x) = x + 1.$$

- Las funciones *proyección* I_n^k de k-dimensiones sobre la coordenada n-ésima,
$$I_n^k(x_1, \ldots, x_k) = x_n,$$
donde $1 \leqslant n \leqslant k$.

Queremos formar la cerradura de la clase de funciones iniciales bajo tres construcciones: composición, recursión primitiva y búsqueda.

Una función h de aridad-k se dice que se obtiene por composición de la función f de aridad-n y las funciones $g_1, \ldots, g_n$ de aridad-k si la ecuación

$$h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_n(\vec{x}))$$

se cumple para todo $\vec{x}$. En el caso de funciones parciales, se entiende aquí que $h(\vec{x})$ está indefinida al menos que $g_1(\vec{x}), \ldots, g_n(\vec{x})$ estén todas definidas y que $\langle g_1(\vec{x}), \ldots, g_n(\vec{x})\rangle$ pertenezca al dominio de f.

Una función h de aridad-$k+1$ se dice que se obtine por *recursión primitiva* de la función f de aridad-k y la función g de aridad-$(k+2)$ (donde $k > 0$) si el par de ecuaciones

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

se cumple para todo $\vec{x}$ y y.

Nuevamente, en el caso de funciones parciales, se entiende que $h(\vec{x}, y+1)$ está indefinida al menos que $h(\vec{x}, y)$ esté definida y que $\langle h(\vec{x}, y), \vec{x}, y\rangle$ esté en el dominio de g.

Observe que en esta situación, conocer las dos funciones f y g determina completamente la función h. Más formalmente, si tanto h_1 como h_2 se obtinen por recursión primitiva de f y g, entonces para cada $\vec{x}$, podemos mostrar por inducción sobre y que $h_1(\vec{x}, y) = h_2(\vec{x}, y)$.

Para el caso $k = 0$, la función h de aridad-uno se obtiene por recursión primitiva de la función g de aridad-dos usando el número m si el par de ecuaciones

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

se cumplen para todo y.

Posponiendo el asunto de la búsqueda, definimos a una función como *recursiva primitiva* si se puede construir de las funciones cero, sucesor y proyección mediante el uso de composición y recursión primitiva. (Ver el principio del Capítulo 2 para algunos ejemplos). En otras palabras, la clase de funciones recursivas primitivas es la clase más pequeña que incluye nuestras funciones iniciales y es cerrada bajo composición y recursión primitiva. (Aquí decir que una clase $\mathcal{C}$ es "cerrada" bajo composición y recursión primitiva significa que siempre que una función f se obtiene por composición de funciones en $\mathcal{C}$ o se obtiene por recursión primitiva de funciones en $\mathcal{C}$, entonces f misma pertenece a $\mathcal{C}$).

Claramente todas las funciones recursivas primitivas son totales. Esto es porque todas las funciones inciales son totales, la composición de funciones totales es total, y una función obtenida por recursión primitiva a partir de funciones totales será total.

Decimos que una relación R sobre $\mathbb{N}$ de aridad-k es recursiva primitiva si su función característica es recursiva primitiva.

Luego uno puede mostrar que una gran cantidad de las funciones comunes sobre $\mathbb{N}$ son recursivas primitivas: adición, multiplicación, ..., la función cuyo valor en m es el $(m+1)$-ésimo primo, ... En el capítulo 2 emprenderemos el proyecto de mostrar que muchas funciones son recursivas primitivas.

Por un lado, parece claro que toda función recursiva primitiva debe considerarse como calculable efectivamente. (Las funciones iniciales son muy

fáciles. La composición no presenta grandes obstáculos. Siempre que h se obtiene por recursión primitiva de f y g calculables efectivamente, entonces vemos que podemos encontrar efectivamente $h(\vec{x}, 99)$, encontrando primero $h(\vec{x}, 0), h(\vec{x}, 1), \ldots, h(\vec{x}, 98)$.) Por otro lado, es posible que la clase de funciones recursivas primitivas no pueda incluir a todas las funciones calculables totales ya que podemos "diagonalizar" la clase.

Esto es, mediante un indexado apropiado del "árbol familiar" de las funciones recursivas primitivas, podemos hacer una lista $f_0, f_1, f_2, \ldots$ de todas las funciones recursivas primitivas de aridad-uno. Luego considere la función diagonal $d(x) = f_x(x) + 1$. Entonces d no puede ser recursiva primitiva; difiere de cada f_x en x. Sin embargo, si hicimos nuestra lista muy pulcramente, la función d será calculable efectivamente. La conclusión es que la clase de funciones recursivas primitivas es una clase extensa pero es una subclase de las funciones calculables totales.

Luego, decimos que una función h de aridad-k se obtiene de una función g de aridad-$(k+1)$ mediante *búsqueda*, y escribimos

$$h(\vec{x}) = \mu\, y[g(\vec{x}, y) = 0]$$

si para cada $\vec{x}$, el valor $h(\vec{x})$ o es el número y tal que $g(\vec{x}, y) = 0$ y $g(\vec{x}, y)$ está definida y no es cero para todo $s < y$, si tal número y existe, sino está indefinida, si tal número y no existe. La idea detrás de este "operador-μ" es la de buscar por el número más pequeño y tal que es la solución a una ecuación, probando sucesivamente con $y = 0, 1, \ldots$.

Obtenemos las funciones *recursivas generales* agregando búsqueda a nuestros métodos de cerradura. Esto es, una función parcial es recursiva general si se puede construir a partir de las funciones cero, sucesor y proyección, usando composición, recursión primitiva y búsqueda (es decir, el operador-μ).

La clase de funciones parciales recursivas generales sobre $\mathbb{N}$ es (como lo demostró Turing) exactamente la misma como la clase de funciones parciales computables a la Turing. Este es un resultado muy llamativo, a la luz de las maneras muy distintas en las que las dos definiciones fueron formuladas. Las máquinas de Turing parecerían, a primera vista, tener poco que ver con recursión primitiva y búsqueda. Y aún así, obtenemos exactamente las mismas funciones parciales a partir de los dos enfoques. Y la tesis de Church, por tanto, tiene la formulación equivalente a que el concepto de función general recursiva es la formalización correcta del concepto informal de calculabilidad efectiva.

¿Qué si tratamos de "diagonalizar" la clase de funciones recursivas generales, como lo hicimos con las funciones recursivas primitivas? Como argumentaremos más tarde, podemos hacer nuevamente una lista ordenada $\phi_0, \phi_1, \phi_2, \ldots$ de todas las funciones parciales reursivas generales. Podemos definir la función diagonal $d(x) = \phi_x(x) + 1$. Pero en esta ecuación, $d(x)$ está indefinida al menos que $\phi_x(x)$ esté definida. La función diagonal d está en realidad entre las funciones parciales recursivas generales, y así es ϕ_k para algún k, pero $d(k)$ debe estar indefinida. Ninguna contradicción resulta.

El uso de la palabra "recursiva" en el contexto de las funciones recursivas primitivas es enteramente razonable. Gödel, escribiendo en Alemán, había usado simplemente "rekursiv" para las funciones recursivas primitivas. (Fue Rózsa Péter quien introdujo el término "recursiva primitiva"). Pero la clase de funciones recursivas generales tiene -como esta sección muestra- otras caracterizaciones en las que la *recursión* juega un papel nada obvio.

Esto nos guía a la pregunta: ¿Cómo llamar a esta clase de funciones? Tener dos nombres ("computable a la Turing" y "recursiva general") es una vergüenza de ricos, y la situación sólo empeorará. Históricamente, el nombre "funciones recursivas parciales" ganó. Y las relaciones sobre $\mathbb{N}$ se dicen *recursivas* si sus funciones características pertenecen a esa clase. El estudio de esas funciones por años fue llamada "teoría de funciones recursivas", y luego "teoría de recursión". Pero esto fue más un accidente histórico que una elección razonada. No obstante, la terminología se ha vuelto estándar.

Pero ahora se está haciendo un esfuerzo para cambiar lo que ha sido la terminología estándar. En consecuencia, este libro, *Teoría de la Computabilidad*, habla de funciones parciales *computables*. Y le llamaremos a una relación *computable* si su función característica es una función computable. Así, el concepto de relación computable corresponde a la noción informal de relación decidible. (El manuscrito de este libro ha, no obstante, sido preparado con macros TEX que facilitarían un cambio rápido de terminología).

En cualquier caso, categóricamente existe la necesidad de tener adjetivos separados para el concepto informal (aquí "calculable" se usa para funciones, y "decidible" para relaciones) y el concepto definido formalmente (aquí "computable").

1.2.3. Programas Loop y While

La idea detrás del concepto de funciones calculables efectivas es que uno debería poder dar instrucciones explícitas (un programa) para calcular dicha

función. ¿Qué lenguaje de programación sería adecuado aquí? En realidad, cualquiera de los lenguajes de programación comúnmente utilizados sería suficiente, si estuviera libre de ciertas limitaciones prácticas, como el tamaño del número denotado por una variable. Damos aquí un lenguaje de programación simple con la propiedad de que las funciones programables son exactamente las funciones parciales computables sobre $\mathbb{N}$.

Las variables del lenguaje son $X_0, X_1, X_2, \ldots$. Aunque hay infinitas variables en el lenguaje, cualquier programa, al ser una cadena finita de comandos, sólo puede tener un número finito de estas variables. Si queremos que el lenguaje esté formado por palabras sobre un alfabeto finito, podemos reemplazar X_3, digamos, por X'''.

Al ejecutar un programa, a cada variable del programa se le asigna un número natural. No hay límite sobre cuán grande puede ser este número. Inicialmente, algunas de las variables contendrán la entrada a la función; el lenguaje no tiene comandos de "entrada". De manera similar, el lenguaje no tiene comandos de "salida"; cuando (y si) el programa se detiene, el valor de X_0 debe ser el valor de la función.

Los comandos del lenguaje son de cinco tipos:

1. $X_n \leftarrow 0$. Este es el comando de *borrado*; su efecto es asignar el valor 0 a X_n.

2. $X_n \leftarrow X_n + 1$. Este es el comando de *incremento*; su efecto es aumentar en uno el valor asignado a X_n.

3. $X_n \leftarrow X_m$. Este es el comando de *copia*; su efecto es justo lo que sugiere el nombre; en particular, deja el valor de X_m sin cambios.

4. loop X_n y endloop X_n. Estos son los comandos *loop* y deben usarse en pares. Es decir, si $\mathcal{P}$ es un programa (una cadena de comandos sintácticamente correcta), entonces también lo es la cadena:

   $$\begin{array}{l} \text{loop } X_n \\ \quad \mathcal{P} \\ \text{endloop } X_n \end{array}$$

   Lo que este programa significa es que $\mathcal{P}$ debe ejecutarse un cierto número k de veces. Y ese número k es el valor inicial de X_n, el valor asignado a X_n antes de comenzar a ejecutar $\mathcal{P}$. Posiblemente $\mathcal{P}$ cambie el valor

de X_n; esto no tiene ningún efecto sobre k. Si $k = 0$, entonces esta cadena no hace nada.

5. while $X_n \neq 0$ y endwhile $X_n \neq 0$. Estos son los comandos *while*; nuevamente, deben usarse en pares, como los comandos de bucle. Pero hay una diferencia. El programa

$$\begin{array}{l} \text{while } X_n \neq 0 \\ \quad \mathcal{P} \\ \text{endwhile } X_n \neq 0 \end{array}$$

también ejecuta el programa $\mathcal{P}$ un número k de veces. Pero ahora k no está determinado de antemano; importa mucho cómo cambia $\mathcal{P}$ el valor de X_n. El número k es el número menor (si lo hay) tal que $\mathcal{P}$ se ejecutará hasta que a X_n se le asigne el valor 0. El programa se ejecutará para siempre si no existe tal k.

Y esos son los únicos comandos. Un programa *while* es una secuencia de comandos, sujeta únicamente al requisito de que el bucle y los comandos while se utilicen en pares, como se ilustra. Claramente, este lenguaje de programación es lo suficientemente simple como para ser simulado por cualquiera de los lenguajes de programación comunes si ignoramos los problemas de desbordamiento.

Un programa *loop* es un programa while sin comandos while; es decir, sólo tiene comandos de borrado, incremento, copia y loop. Tenga en cuenta la propiedad importante: un programa loop *siempre se detiene*, pase lo que pase. Pero es fácil hacer un programa while que nunca se detenga.

Decimos que una función parcial f de aridad k sobre $\mathbb{N}$ es *computable-while* si existe un programa while $\mathcal{P}$ que, siempre que comience con una tupla $\vec{x}$ de aridad k asignada a las variables $X_1, \ldots, X_k$ y 0 asignado al resto de variables, se comporta de la siguiente manera:

- Si $f(\vec{x})$ está definida, entonces el programa eventualmente se detiene, y X_0 contiene el valor de $f(\vec{x})$.
- Si $f(\vec{x})$ no está definida, entonces el programa nunca se detiene.

Las funciones *computables-loop* se definen de forma análoga. Pero existe la diferencia de que cualquier función computable-loop es total.

Teorema 1.2.2

(a) Una *función* sobre $\mathbb{N}$ es *computable-loop* si y sólo si es *recursiva primitiva*.

(b) Una *función parcial* sobre $\mathbb{N}$ es *computable-while* si y sólo si es *recursiva general*.

La prueba en una dirección, para demostrar que toda función recursiva primitiva es computable-loop, implica una serie de ejercicios de programación. La prueba en la otra dirección implica codificar el estado de un programa $\mathcal{P}$ sobre la entrada $\vec{x}$ después de t pasos, y mostrar que existen funciones recursivas primitivas que nos permiten determinar el estado después de $t+1$ pasos y el estado terminal.

Debido a que la clase de funciones parciales recursivas generales coincide con la clase de funciones parciales computables de Turing, podemos concluir del teorema anterior que la computabilidad while coincide con la computabilidad de Turing.

1.2.4. Máquinas de Registros

Aquí hay otro lenguaje de programación. Por un lado, es extremadamente simple, incluso más simple que el lenguaje de los programas loop-while. Por otro lado, el lenguaje es "desestructurado"; incorpora (de hecho) comandos go-to. Esta formalización fue presentada por Shepherdson y Sturgis en un artículo de 1963.

Una *máquina de registros* debe considerarse como un dispositivo de cómputo con un número finito de "registros", numerados $0, 1, 2, \ldots, K$. Cada registro es capaz de almacenar un número natural de cualquier magnitud; no hay límite para el tamaño de este número. El funcionamiento de la máquina está determinado por un programa. Un programa es una secuencia finita de instrucciones, extraídas de la siguiente lista:

- "Incrementa r", $I\ r$ (donde $0 \leqslant r \leqslant K$): El efecto de esta instrucción es aumentar el contenido del registro r en 1. Luego, la máquina pasa a la siguiente instrucción del programa (si corresponde).

- "Decrementa r", $D\ r$ (donde $0 \leqslant r \leqslant K$): El efecto de esta instrucción depende del contenido del registro r. Si ese número es distinto de cero, se decrementa en 1 y la máquina no pasa a la siguiente instrucción, sino

a la siguiente de la siguiente. Pero si el número en el registro r es cero, la máquina simplemente pasa a la siguiente instrucción. En resumen, la máquina intenta disminuir el registro r y, si tiene éxito, se salta una instrucción.

- "Jump q", $J\ q$ (donde q es un número entero: positivo, negativo o cero): todos los registros se dejan sin cambios. La máquina toma como siguiente instrucción la $q-$ésima instrucción que sigue a ésta en el programa (si $q \geqslant 0$), o la $|q|-$ésima instrucción que precede a ésta (si $q < 0$). La máquina se detiene si no existe tal instrucción en el programa. La instrucción $J\ 0$ da como resultado un bucle, en el que la máquina ejecuta esta instrucción una y otra vez.

Y eso es todo. El lenguaje tiene sólo estos tres tipos de instrucciones. (Estrictamente hablando, en estas instrucciones, r y q son numerales, no números. Es decir, una instrucción debe ser una secuencia de símbolos. Si usamos números en base 10, entonces el alfabeto es $\{I, D, J, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -\}$. Una instrucción es una palabra correctamente formada sobre este alfabeto). **Ejemplos:**

1. CLEAR 7: un programa para borrar el registro 7.

$D\ 7$	Trata de decrementar el registro 7.
$J\ 2$	Para.
$J\ -2$	Regresa y repite.

2. MOVE from r to s: un programa para mover un número del registro r al registro s (donde $r \neq s$).

CLEAR s	Use el programa del primer ejemplo.
$D\ r$	Trata de decrementar el registro r.
$J\ 3$	Para cuando sea cero.
$I\ s$	incrementa el registro s.
$J - 3$	Regresa y repite.

3. ADD 1 to 2 and 3: un programa para sumar el contenido del registro

1 a los registros 2 y 3.

$$D\ 1$$
$$J\ 4$$
$$I\ 2$$
$$I\ 3$$
$$J-4$$

4. COPY from r to s: un programa para copiar un número del registro r al registro s (dejando el registro r sin cambios). Combinamos los ejemplos anteriores.

CLEAR s	Use el programa del primer ejemplo.
MOVE from r to s	Use el programa del segundo ejemplo.
ADD t to r and s	Use el programa del tercer ejemplo.

   Este programa tiene 15 instrucciones. Utiliza un tercer registro, el registro t. Al final, se restaura el contenido del registro r. Pero durante la ejecución, se debe borrar el registro r; ésta es la única manera de determinar su contenido. (Aquí se supone que r, s y t son distintos).

5. (Adición) Digamos que x y y están en los registros 1 y 2. Queremos $x + y$ en el registro 0, y queremos dejar x y y todavía en los registros 1 y 2 al final.

	Contenido de los registros			
CLEAR 0	0	x	y	
MOVE from 1 to 3	0	0	y	x
ADD 3 to 1 and 0	x	x	y	0
MOVE from 2 to 3	x	x	0	y
ADD 3 to 2 and 0	$x+y$	x	y	0

   Este programa tiene 27 instrucciones tal como está escrito, pero tres de ellas son innecesarias. (En la cuarta línea, comenzamos borrando el registro 3, que ya está limpio).

Ahora supongamos que f es una función parcial de aridad n sobre $\mathbb{N}$. Posiblemente, habrá un programa $\mathcal{P}$ tal que si arrancamos una máquina de registros (que tiene todos los registros a los que se refiere $\mathcal{P}$) con $x_1, \ldots, x_n$ en los registros $1, \ldots, n$ y 0 en los otros registros, y aplicamos el programa $\mathcal{P}$, entonces se cumplen las siguientes condiciones:

- Si $f(x_1, \ldots, x_n)$ está definida, entonces el cálculo finalmente termina con $f(x_1, \ldots, x_n)$ en el registro 0. Además, el cálculo termina al llegar a la $(p+1)$-ésima instrucción, donde p es la longitud de $\mathcal{P}$.

- Si $f(x_1, \ldots, x_n)$ no está definida, el cálculo nunca termina.

Si existe tal programa $\mathcal{P}$, decimos que $\mathcal{P}$ calcula f.

¿Qué funciones son computables por programas de máquinas de registros? El lenguaje es tan simple (parece un lenguaje de juguete) que la primera impresión podría ser que sólo funciones muy simples son computables. Esta impresión es engañosa.

Teorema 1.2.3 Sea f una función parcial. Entonces, existe un programa de máquinas de registros que calcula f si y sólo si f es una función parcial recursiva general.

Así, al utilizar máquinas de registros, llegamos exactamente a la clase de funciones parciales recursivas generales, una clase que definimos originalmente en términos de recursión primitiva y búsqueda.

1.2.5. Definibilidad en lenguajes formales

Esbozaremos brevemente otras formas en las que podría formalizarse el concepto de calculabilidad efectiva. Los detalles quedarán a la imaginación.

En 1936, en su artículo en el que presentaba lo que ahora se conoce como la tesis de Church, Alonzo Church utilizó un sistema formal, el *cálculo*$-\lambda$. Church había desarrollado este sistema como parte de su estudio de los fundamentos de la lógica. En particular, para cada número natural, n, existe una fórmula $\bar{n}$ del sistema que denota n, es decir, un numeral para n. Más importante aún, se podrían utilizar fórmulas para representar la construcción de funciones. Definió una función F de aridad dos como *definible*$-\lambda$ si existía una fórmula F del cálculo lambda tal que siempre que $F(m, n) = r$, entonces la fórmula $\{\boldsymbol{F}\}(\bar{m}, \bar{n})$ fuera convertible, siguiendo las reglas del sistema, a la fórmula $\bar{r}$. Esta definición se puede extender a funciones de aridad k.

Stephen Kleene, que fue alumno de Church, demostró que una función era definible$-\lambda$ si y sólo si era recursiva general. (Church y su alumno J. B. Rosser también participaron en el desarrollo de este resultado). Church escribió en su artículo: "El hecho. . . de que dos definiciones tan diferentes y (en opinión del autor) igualmente naturales de calculabilidad efectiva resulten

ser equivalentes aumenta la fuerza de las razones. . . por creer que constituyen una caracterización tan general de esta noción como es consistente con la comprensión intuitiva habitual de la misma".

Anteriormente, en 1934, Kurt Godel, en conferencias en Princeton, formuló un concepto que ahora se conoce como computabilidad de Godel-Herbrand. Sin embargo, en ese momento no propuso el concepto como una formalización del concepto de calculabilidad efectiva. El concepto implicaba un cálculo formal de ecuaciones entre términos construidos a partir de variables y símbolos de funciones. El cálculo permitía pasar de una ecuación $A = B$ a otra ecuación obtenida sustituyendo una parte C de A o B por otro término D del que se había derivado la ecuación $C = D$. Si un conjunto $\mathcal{E}$ de ecuaciones permitía derivar, en un sentido adecuado, exactamente los valores correctos para una función f sobre $\mathbb{N}$, entonces se decía que $\mathcal{E}$ era un conjunto de ecuaciones recursivas para f. Una vez más, resultó que existía un conjunto de ecuaciones recursivas para f si y sólo si f era una función recursiva general.

Un enfoque bastante diferente para caracterizar las funciones efectivamente calculables implica la definibilidad mediante expresiones en lógica simbólica. Un lenguaje formal para la aritmética de números naturales podría tener variables y un número para cada número natural, y símbolos para la relación de igualdad y, al menos, para las operaciones de suma y multiplicación. Además, el lenguaje debería poder manejar los conectivos lógicos básicos como "y", "o" y "no". Finalmente, debe incluir las expresiones "cuantificadoras" $\forall v$ y $\exists v$ que significan "para todos los números naturales v" y "para algún número natural v", respectivamente.

Por ejemplo,

$$\exists s(u_1 + s = u_2)$$

podría ser una expresión en el lenguaje formal, afirmando una propiedad de u_1 y u_2. La expresión es verdadera (en $\mathbb{N}$ con sus operaciones habituales) cuando a u_1 se le asigna 4 y a u_2 se le asigna 9 (tome $s = 5$). Pero es falso cuando a u_1 se le asigna 9 y a u_2 se le asigna 4. De manera más general, podemos decir que la expresión define (en $\mathbb{N}$ con sus operaciones habituales) la relación binaria "$\leqslant$" sobre $\mathbb{N}$.

Para otro ejemplo,

$$v \neq 0 \wedge \forall x \forall y[\exists s(v + s = y) \vee \exists t(v + t = y) \vee v \neq x \cdot y]$$

podría ser una expresión en el lenguaje formal, afirmando una propiedad de v. La expresión es falsa (sobre $\mathbb{N}$ con su operación habitual) cuando a v se le

asigna el número 6 (pruebe con $x = 2$ y $y = 3$). Pero la expresión es verdadera cuando a v se le asigna 7. De manera más general, la expresión es verdadera cuando a v se le asigna un número primo, y sólo entonces. Podemos decir que esta expresión define el conjunto de los números primos (sobre $\mathbb{N}$ con sus operaciones habituales).

Decimos que una función parcial f de aridad k sobre $\mathbb{N}$ es *definible*$-\Sigma_1$ si la gráfica de f (es decir, la relación de aridad $(k+1)$ $\{< \vec{x}, y > | f(\vec{x}) = y\}$) puede ser definida sobre $\mathbb{N}$ con las operaciones de suma, multiplicación y exponenciación, mediante una expresión de la siguiente forma:

$$\exists v_1 \exists v_2 \cdots \exists v_n (\text{expresión sin cuantificadores})$$

Entonces la clase de funciones parciales *definibles*$-\Sigma_1$ coincide exactamente con la clase de funciones parciales dada por las otras formalizaciones de calculabilidad descritas aquí. Además, Yuri Matiyasevich demostró en 1970 que aquí no era necesaria la operación de exponenciación.

Finalmente, digamos que una función parcial f de aridad k sobre $\mathbb{N}$ es *representable* si existe alguna teoría finitamente axiomatizable $\boldsymbol{T}$ en un lenguaje que tenga un numeral adecuado $\bar{n}$ para cada número natural n, y existe una fórmula φ de ese lenguaje tal que (para cualquier número natural) $f(x_1, \ldots, x_k) = y$ si y sólo si $\varphi(\bar{x_1}, \ldots, \bar{x_k}, \bar{y})$ es una oración deducible en la teoría $\boldsymbol{T}$. Entonces, una vez más, la clase de funciones parciales representables coincide exactamente con la clase de funciones parciales dada por las otras formalizaciones de calculabilidad descritas aquí.

1.2.6. La tesis de Church revisada

En resumen, para una función parcial f de aridad k, las siguientes condiciones son equivalentes:

- La función f es una función parcial Turing-computable.
- La función f es una función parcial recursiva general.
- La función parcial f es computable-while.
- La función parcial f se calcula mediante algún programa de máquinas de registros.
- La función parcial f es definible$-\lambda$.

- La función parcial f es definible$-\Sigma_1$ (sobre los números naturales con suma, multiplicación y exponenciación).
- La función parcial f es representable (en alguna teoría finitamente axiomatizable).

¡La equivalencia de estas condiciones es seguramente un hecho notable! Además, es evidencia de que las condiciones caracterizan alguna propiedad natural y significativa. La tesis de Church es la afirmación de que las condiciones, de hecho, capturan el concepto informal de una función efectivamente calculable.

Definición 1.2.1 Una función parcial f de aridad k sobre los números naturales se dice que es una función parcial computable si se cumplen las condiciones anteriores.

Entonces la tesis de Church es la afirmación de que esta definición es la que queremos.

La situación es algo análoga a la del cálculo. Una función intuitivamente continua (definida en un intervalo) es aquella cuya gráfica se puede dibujar sin levantar el lápiz del papel. Pero para demostrar teoremas se necesita alguna contraparte formalizada de este concepto. Y así se da la definición habitual de continuidad $\epsilon - \delta$. Entonces es justo preguntar si el concepto preciso de continuidad $\epsilon - \delta$ es una formalización precisa de la continuidad intuitiva. En todo caso, la clase de funciones continuas $\epsilon - \delta$ es demasiado amplia. Incluye funciones no diferenciables en ninguna parte, cuyas gráficas no se pueden dibujar sin levantar el lápiz; no hay forma de impartir un vector de velocidad al lápiz. Pero sea exacta o no, se ha descubierto que la clase de funciones continuas $\epsilon - \delta$ es una clase natural e importante en el análisis matemático.

La misma situación ocurre con la computabilidad. Es justo preguntarse si el concepto preciso de función parcial computable es una formalización precisa del concepto informal de función efectivamente calculable. Una vez más, la clase definida con precisión parece ser, en todo caso, demasiado amplia, porque incluye funciones que requieren, para grandes entradas, cantidades absurdas de tiempo de cálculo. La computabilidad corresponde a la calculabilidad efectiva en un mundo idealizado, donde se ignoran la duración del cálculo y la cantidad de espacio de memoria. Pero en cualquier caso, se ha descubierto que la clase de funciones parciales computables es una clase natural e importante.

1.2.7. Ejercicios

1. De un programa-loop para calcular la siguiente función:

$$f(x,y,z) = \begin{cases} y & \text{si } x = 0 \\ z & \text{si } x \neq 0 \end{cases}$$

2. Sea $x \dot{-} y = max(x - y, 0)$, el resultado de restar y de x, pero con un "piso" de 0. Proporcione un programa loop que calcule la función de aridad dos $x \dot{-} y$.

3. Proporcione un programa loop que cuando se inicia con todas las variables asignadas a 0, se detiene con X_0 asignado algún número mayor que 1000.

4. Proporcione un programa de máquinas de registros que calcule la función de resta, $x \dot{-} y = max(x - y, 0)$, como en el ejercicio 2.

5. Proporcione un programa de máquinas de registros que calcule la función parcial de resta:

$$f(x,y) = \begin{cases} x - y & \text{si } x \geqslant y \\ \uparrow & \text{si } x < y \end{cases}$$

6. Proporcione un programa de máquinas de registros que calcule la función de multiplicación, $x \cdot y$.

7. Proporcione un programa de máquinas de registros que calcule la función $\max(x, y)$.

8. Proporcione un programa de máquinas de registros que calcule la función de paridad:

$$f(x) = \begin{cases} 1 & \text{si } x \text{ es impar} \\ 0 & \text{si } x \text{ es par} \end{cases}$$

Capítulo 2

Funciones recursivas generales

En el capítulo anterior vimos una visión general de varias formalizaciones posibles del concepto de calculabilidad efectiva. En este capítulo, nos centramos en una de ellas: la recursividad primitiva y la búsqueda, que nos dan la clase de funciones parciales recursivas generales. En particular, desarrollamos herramientas para mostrar que ciertas funciones están en esta clase. Estas herramientas se utilizarán en el Capítulo 3, donde estudiaremos la computabilidad mediante programas de máquinas de registros.

2.1. Funciones recursivas primitivas

Las funciones recursivas primitivas se han definido en el capítulo anterior como las funciones sobre $\mathbb{N}$ que se pueden construir a partir de funciones cero

$$f(x_1, \ldots, x_k) = 0.$$

la función sucesor

$$S(x) = x + 1.$$

las funciones proyección

$$I_n^k(x_1, \ldots, x_k) = x_n,$$

usando (cero o más veces) composición

$$h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_n(\vec{x}))$$

y recursión primitiva

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

donde $\vec{x}$ puede ser vacía

$$h(0) = m$$
$$h(y+1) = g(h(y), y)$$

Ejemplo 2.1.1 Supongamos que nos dan el número $m = 1$ y la función $g(w, y) = w \cdot (y + 1)$. Entonces la función h obtenida por recursión primitiva de g usando m es la función dada por el par de ecuaciones

$$h(0) = m = 1$$
$$h(y+1) = g(h(y), y) = h(y) \cdot (y + 1).$$

Usando este par de ecuaciones, podemos proceder a calcular los valores de la función h:

$$h(0) = m = 1$$
$$h(1) = g(h(0), 0) = g(1, 0) = 1$$
$$h(2) = g(h(1), 1) = g(1, 1) = 2$$
$$h(3) = g(h(2), 2) = g(2, 2) = 6$$
$$h(4) = g(h(3), 3) = g(6, 3) = 24$$

Etcétera. Para calcular $h(4)$, primero necesitamos saber $h(3)$, y encontrar que necesitamos $h(2)$, y así sucesivamente. La función h en este ejemplo es, por supuesto, más conocida como función factorial, $h(x) = x!$.

Debería quedar bastante claro que dado cualquier número m y cualquier función g de aridad dos, existe una función única h obtenida por recursión primitiva de g usando m. Es la función h la que calculamos como en el ejemplo anterior. De manera similar, dada una función f de aridad k y una función g de aridad $(k + 2)$, existe una función h única de aridad $(k + 1)$ que se obtiene

mediante recursión primitiva de f y g. Es decir, h es la función dada por el par de ecuaciones

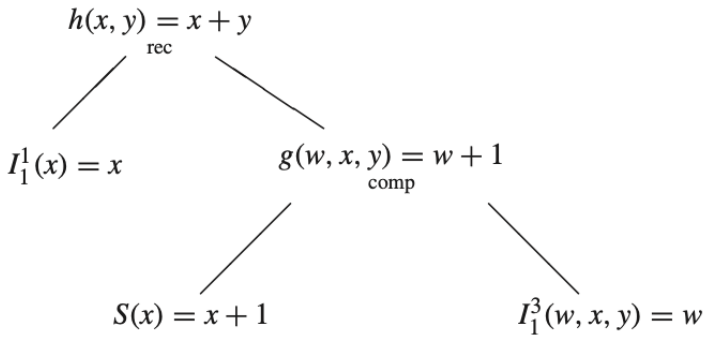
$$h(\vec{x}, 0) = f(\vec{x})$$
$$h(\vec{x}, y+1) = g(h(\vec{x}, y), \vec{x}, y).$$

Además, si f y g son funciones totales, entonces h también será total.

Ejemplo 2.1.2 Considere la función de suma $h(x, y) = x+y$. Para cualquier x fija, su valor en $y+1$ (es decir, $x+y+1$) se puede obtener a partir de su valor en y (es decir, $x+y$) con el simple paso de sumar uno:

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

Este par de ecuaciones muestra que la suma se obtiene mediante recursividad primitiva a partir de las funciones $f(x) = x$ y $g(w, x, y) = w + 1$. Estas funciones f y g son recursivas primitivas; f es la función de proyección I_1^1 y g se obtiene por composición de la función sucesor y I_1^3. Al juntar estas observaciones, podemos formar un árbol que muestra cómo se construye la suma a partir de las funciones iniciales mediante composición y recursividad primitiva:



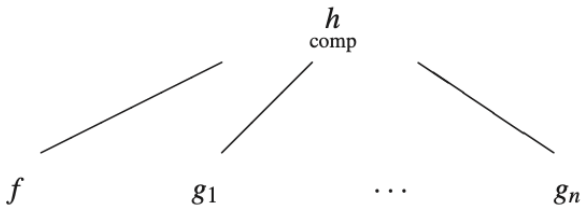
De manera más general, para cualquier función recursiva primitiva h, podemos usar un árbol etiquetado ("árbol de construcción") para ilustrar exactamente cómo se construye h, como en el ejemplo de la suma. En el vértice superior (raíz), colocamos h. En cada vértice minimal (una hoja), tenemos una función inicial: la función sucesora, una función cero o una

función de proyección. En cada uno de los demás vértices, mostramos una aplicación de composición o una aplicación de recursividad primitiva.

Una aplicación de composición.

$$h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_n(\vec{x}))$$

se puede ilustrar en un árbol mediante un vértice con $(n+1)$ ramificaciones:

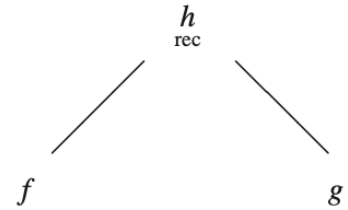


Aquí f debe ser una función de aridad n y $g_1, \ldots, g_n$ todas deben tener la misma aridad que h.

Una aplicación de recursividad primitiva para obtener una función h de aridad $(k+1)$

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

Puede ser ilustrada por un vértice con ramificación binaria:



Tenga en cuenta que si f es de aridad k entoces g debe tener aridad $(k+2)$ y h tendrá aridad $(k+1)$, como ya se había mencionado (por ejemplo, si h es una función de aridad dos, entonces g debe ser una función de aridad tres y f debe ser una función de aridad uno).

El caso $k = 0$, donde una función h de aridad uno se obtiene mediante recursividad primitiva a partir de una función g de aridad dos usando el

número m

$$h(m) = m$$
$$h(x+1) = g(h(x), x),$$

Puede ser ilustrado por un vértice con ramificación unaria.

En ambas formas de recursividad primitiva ($k > 0$ y $k = 0$), la característica clave es que el valor de la función en un número $t + 1$ se puede obtener de alguna manera a partir de su valor en t. El papel de g es explicar cómo.

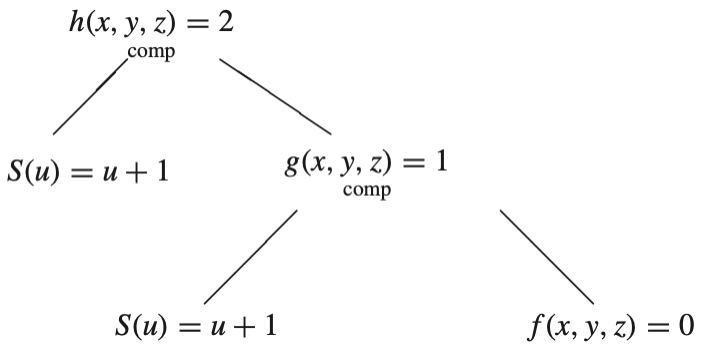
Toda función recursiva primitiva es total. Podemos ver esto por "inducción estructural". Para la base, todas las funciones iniciales (las funciones cero, la función sucesor y las funciones de proyección) son totales. Para los dos pasos inductivos, observamos que la composición de funciones totales produce una función total, y la recursión primitiva aplicada a funciones totales produce una función total. Entonces, para cualquier función recursiva primitiva, podemos avanzar hasta su árbol de construcción. En las hojas del árbol tenemos funciones totales. Y cada vez que nos movemos a un vértice superior, todavía tenemos una función total. Finalmente, llegamos a la raíz superior y concluimos que la función que se está construyendo es total.

A continuación queremos crear un catálogo de funciones recursivas primitivas básicas. Estos elementos del catálogo se pueden utilizar luego como piezas listas para usar para la posterior creación de otras funciones recursivas primitivas.

1. Ya se ha demostrado que la suma $< x, y > \mapsto x + y$ es recursiva primitiva. El símbolo "$\mapsto$" se lee "se mapea a". El símbolo nos brinda una forma muy conveniente de nombrar funciones. Por ejemplo, la función de elevar al cuadrado puede denominarse mediante la frase larga "la función que dado un número, lo eleva al cuadrado". Es matemáticamente conveniente utilizar una letra (como x o t). Esto nos lleva a los

nombres "la función cuyo valor en x es x^2 " o "la función cuyo valor en t es t^2". De manera más compacta, estos nombres se pueden escribir en símbolos como "$x \mapsto x^2$" o "$t \mapsto t^2$". La letra x o t es una variable ficticia; podemos usar cualquier letra aquí.

2. Cualquier función constante $\vec{x} \mapsto k$ se puede obtener aplicando composición k veces a la función sucesora y a la función cero $\vec{x} \mapsto 0$. Por ejemplo, la función de aridad tres que constantemente toma el valor 2 se puede construir mediante el siguiente árbol:



3. Para la multiplicación $< x, y > \mapsto x \times y$, primero observamos que

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

   Esto muestra que la multiplicación se obtiene mediante recursión primitiva de las funciones $x \mapsto 0$ y $< w, x, y > \mapsto w + x$. Esta última función se obtiene mediante composición aplicada a funciones de suma y proyección.

   Ahora podemos concluir que cualquier función polinomial con coeficientes positivos es recursiva primitiva. Por ejemplo, podemos ver que la función $p(x, y) = x^2y + 5xy + 3y^3$ es recursiva primitiva al aplicar repetidamente 1, 2 y 3.

4. La exponenciación $< x, y > \mapsto x^y$ es similar:

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

5. La exponenciación $< x, y > \mapsto y^x$ se obtiene de la función anterior mediante composición con funciones de proyección. (Las funciones de los puntos 4 y 5 son funciones diferentes; asignan valores diferentes a $< 2, 3 >$. El hecho de que coincidan en $< 2, 4 >$ es un accidente). Deberíamos generalizar esta observación. Por ejemplo, si f es recursiva primitiva y g está definido por la ecuación

$$g(x, y, z) = f(y, 3, x, x)$$

entonces g también es recursiva primitiva, y se obtiene por composición a partir de f y funciones de proyección y constantes. Diremos en esta situación que g se obtiene de f mediante *transformación explícita*. La transformación explícita permite cambiar variables, repetir variables, omitir variables y sustituir constantes.

6. La función factorial $x!$ satisface el par de ecuaciones recursivas

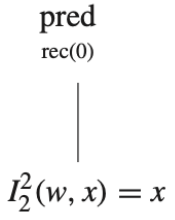
$$0! = 1$$
$$(x + 1)! = x! \times (x + 1).$$

De este par de ecuaciones se deduce que la función factorial se obtiene mediante recursividad primitiva (usando 1) de la función $g(w, x) = w \cdot (x + 1)$. (Vea el ejemplo al comienzo de este capítulo).

7. La función predecesor $pred(x) = x - 1$ (excepto que $pred(0) = 0$) se obtiene mediante recursión primitiva de I_2^2

$$pred(0) = 0$$
$$pred(x + 1) = x.$$

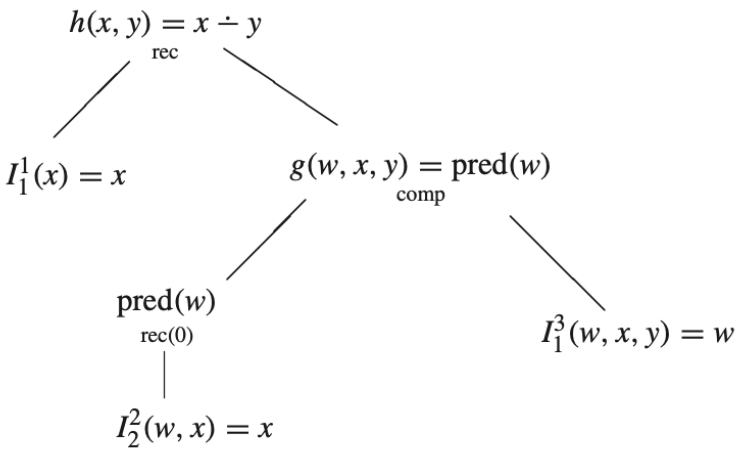
Este par de ecuaciones conduce al árbol:



8. Defina la función de resta propia $x \dot{-} y$ mediante la ecuación $x \dot{-} y = max(x - y, 0)$. Esta función es recursiva primitiva:

$$x \dot{-} 0 = x$$
$$x \dot{-} (y + 1) = pred(x \dot{-} y)$$

Este par de ecuaciones de recursividad produce el siguiente árbol de construcción:



Por cierto, el símbolo $\dot{-}$ a veces se lee como "monus".

9. Supongamos que f es recursiva primitiva y defina las funciones s y p mediante las ecuaciones

$$s(\vec{x}, y) = \sum_{t<y} f(\vec{x}, t) \quad \text{y} \quad p(\vec{x}, y) = \prod_{t<y} f(\vec{x}, t)$$

(sujeto a las convenciones estándar para la suma vacía $\sum_{t<0} f(\vec{x}, t) = 0$ y el producto vacío $\prod_{t<0} f(\vec{x}, t) = 1$). Entonces tanto s como p son recursivas primitivas. Para p, tenemos el par de ecuaciones:

$$p(\vec{x}, 0) = 1$$
$$p(\vec{x}, y + 1) = p(\vec{x}, y) \cdot f(\vec{x}, y)$$

10. Defina la función z mediante la ecuación.

$$z(x) = \begin{cases} 1 & \text{si } x = 0 \\ 0 & \text{si } x > 0. \end{cases}$$

Es decir, la función z busca si su entrada es cero y devuelve Sí (es decir, 1) si es cero; de lo contrario, devuelve No (es decir, 0). La función z es recursiva primitiva. Podemos ver esto en la ecuación $z(x) = 0^x$. Más directamente, podemos verlo en la ecuación $z(x) = 1 \dot{-} x$. Y aún más directamente, podemos verlo en las ecuaciones de recursividad.

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

mostrando que z se obtiene mediante recursividad primitiva (usando 1) de la función $g(w, x) = 0$.

$$\begin{array}{c} z \\ \text{rec(1)} \\ | \\ g(w,x) = 0 \end{array}$$

11. De manera similar, la función h que verifica sus dos entradas x y y para ver si o no $x \leqslant y$

$$h(x,y) = \begin{cases} 1 & \text{si } x \leqslant y \\ 0 & \text{en caso contrario} \end{cases}$$

es recursiva primitiva porque $h(x, y) = z(x \dot{-} y)$.

Los ítems 10 y 11 pueden reformularse en términos de relaciones (en lugar de funciones). Supongamos que R es una relación de aridad k sobre los números naturales, es decir, R es algún conjunto de tuplas-k de números naturales: $R \subseteq N^k$. Definimos R como una relación recursiva primitiva si su función característica

$$C_R(x_1, \ldots, x_k) = \begin{cases} 1 & \text{si } (x_1, \ldots, x_k) \in R \\ 0 & \text{en caso contrario} \end{cases}$$

es una función recursiva primitiva. Por ejemplo, el item 11 establece que la relación de ordenamiento $\{< x, y > | x \leqslant y\}$ es una relación binaria

recursiva primitiva. Y el item 10 establece que $\{0\}$ es una relación unaria recursiva primitiva.

De la composición derivamos la regla de sustitución: si Q es una relación recursiva primitiva de aridad n, y $g_1, \ldots, g_n$ son funciones recursivas primitivas de aridada k, entonces la relación de aridad k.

$$\{\vec{x}|g_1(\vec{x}), \ldots, g_n(\vec{x})\}$$

es recursiva primitiva porque su función característica se obtiene de C_Q y $g_1, \ldots, g_n$ por composición.

De una relación Q, podemos formar su *complemento* $\overline{Q}$:

$$\overline{Q} = \{\vec{x}|\vec{x} \text{ no está en } Q\}.$$

A partir de dos relaciones Q y R de aridad k (para el mismo k), podemos formar su *intersección*,

$$Q \cap R = \{\vec{x}| \text{ ambos } \vec{x} \in Q \text{ y } \vec{x} \in R\}$$

y su *unión*

$$Q \cup R = \{\vec{x}| \text{ o } \vec{x} \in Q \text{ o } \vec{x} \in R \text{ o ambos}\}$$

Podemos simplificar ligeramente la notación escribiendo, en lugar de $\vec{x} \in Q$, simplemente $Q(\vec{x})$. En esta notación,

$$\begin{aligned} \overline{Q} &= \{\vec{x}| \text{ no } Q(\vec{x})\}, \\ Q \cap R &= \{\vec{x}| \text{ ambos } Q(\vec{x}) \text{ y } R(\vec{x})\}, \\ Q \cup R &= \{\vec{x}| \text{ o } Q(\vec{x}) \text{ o } R(\vec{x}) \text{ o ambos}\}. \end{aligned}$$

El siguiente teorema nos asegura que estas construcciones preservan la recursividad primitiva. Es decir, cuando se aplican a relaciones recursivas primitivas, producen relaciones recursivas primitivas. Este teorema será útil para ampliar nuestro catálogo de relaciones y funciones recursivas primitivas.

Teorema *Supongamos que Q y R son relaciones recursivas primitivas de aridad k. Entonces las siguientes relaciones también son recursivas primitivas.*

(a) *El complemento* $\overline{Q}$ *de* Q*:*

$$\overline{Q} = \{\vec{x}| \text{ no } Q(\vec{x})\}$$

(b) *La intersección* $Q \cap R$ *de* Q *y* R*:*

$$Q \cap R = \{\vec{x}| \text{ ambos } Q(\vec{x}) \text{ y } R(\vec{x})\}$$

(c) *La unión* $Q \cup R$ *de* Q *y* R*:*

$$Q \cup R = \{\vec{x}| \text{ o } Q(\vec{x}) \text{ o } R(\vec{x}) \text{ o ambos}\}$$

Prueba:

(a)
$$C_{\overline{Q}}(\vec{x}) = z(C_Q(\vec{x}))$$
donde z es la función del item 10. Es decir, $C_{\overline{Q}}$ se obtiene por composición de funciones que se sabe que son recursivas primitivas. Las demás partes se prueban de manera similar; necesitamos hacer la función característica a partir de partes recursivas primitivas.

(b)
$$C_{Q\cap R}(\vec{x}) = C_Q(\vec{x}) \cdot C_R(\vec{x})$$

(c)
$$C_{Q\cup R}(\vec{x}) = \text{pos}[C_Q(\vec{x}) + C_R(\vec{x})]$$
donde pos es la función del Ejercicio 5 de este capítulo. $\dashv$

Por ejemplo, podemos aplicar este teorema para concluir que $>$ y $=$ son relaciones recursivas primitivas:

12. La relación $\{< x, y > | x > y\}$ es recursiva primitiva porque es el complemento de la relación $\leqslant$ del item 11.

13. La relación $\{< x, y > | x = y\}$ es recursiva primitiva porque es la intersección de las relaciones $\leqslant$ y $\geqslant$, y $\geqslant$ se obtiene de $\leqslant$ mediante transformación explícita. Del item 13 y de la regla de sustitución se deduce que para cualquier función recursiva primitiva f, su gráfica

$$\{< \vec{x}, y > | f(\vec{x}) = y\}$$

es una relación recursiva primitiva.

Definición por casos: *Supongamos que Q es una relación recursiva primitiva de aridad k, y que f y g son funciones recursivas primitivas de aridad k. Entonces la función h definida por la ecuación*

$$h(\vec{x}) = \begin{cases} f(\vec{x}) & \text{si } Q(\vec{x}) \\ g(\vec{x}) & \text{si no } Q(\vec{x}) \end{cases}$$

también es recursiva primitiva.

Prueba: $h(\vec{x}) = f(\vec{x}) \cdot C_Q(\vec{x}) + g(\vec{x}) \cdot C_{\overline{Q}}(\vec{x})$. $\dashv$

Este resultado puede extenderse a más de dos casos; vea el Ejercicio 12 de este capítulo. Por ejemplo, es posible que deseemos manejar una ecuación de la forma

$$h(\vec{x}) = \begin{cases} f_1(\vec{x}) & \text{si } Q(\vec{x}) \text{ y } R(\vec{x}) \\ f_2(\vec{x}) & \text{si } Q(\vec{x}) \text{ y no } R(\vec{x}) \\ f_3(\vec{x}) & \text{si } R(\vec{x}) \text{ y no } Q(\vec{x}) \\ f_4(\vec{x}) & \text{si ni } Q(\vec{x}) \text{ ni } R(\vec{x}) \end{cases}$$

o una de la forma

$$h(\vec{x}) = \begin{cases} f_1(\vec{x}) & \text{si } Q_1(\vec{x}) \\ f_2(\vec{x}) & \text{si } Q_2(\vec{x}) \\ \cdots & \cdots \\ f_9(\vec{x}) & \text{si } Q_9(\vec{x}) \\ f_{10}(\vec{x}) & \text{si ninguna de las anteriores se cumple} \end{cases}$$

en una situación en la que se sabe que no hay dos de $Q_1, \ldots, Q_9$ que puedan cumplirse simultáneamente.

Además, a partir de una relación Q de aridad k, podemos formar

$$\{< x_1, \ldots, x_{k-1}, y > \mid \text{para todo } t < y, < x_1, \ldots, x_{k-1}, t > \in Q\}$$

que se puede escribir en mejor notación como

$$\{< \vec{x}, y > \mid (\forall t < y)\ Q(\vec{x}, t)\}$$

donde el símbolo $\forall$ se lee "para todos". Con el mismo espíritu, podemos formar

$$\{< x_1, \ldots, x_{k-1}, y > \mid \text{para algún } t < y, < x_1, \ldots, x_{k-1}, t > \in Q\}$$

que está mejor escrito como

$$\{< \vec{x}, y > \mid (\exists t < y)\ Q(\vec{x}, t)\}$$

donde el símbolo $\exists$ se lee como "existe ... tal que".

Nuevamente, estas construcciones preservan la recursividad primitiva:
Teorema: *Supongamos que Q es una relación recursiva primitiva de aridad $(k+1)$. Entonces las siguientes relaciones también son recursivas primitivas:*

(a)
$$\{< \vec{x}, y > \mid (\forall t < y)\ Q(\vec{x}, t)\}$$

(b)
$$\{< \vec{x}, y > \mid (\exists t < y)\ Q(\vec{x}, t)\}$$

Prueba:

(a) El valor de la función característica en $< \vec{x}, y >$ es

$$\prod_{t<y} C_Q(\vec{x}, t)$$

Aplicar el item 9.

(b) El valor de la función característica en $< \vec{x}, y >$ es

$$\text{pos}\left[\sum_{t<y} C_Q(\vec{x}, t)\right]$$

donde pos es la función del ejercicio 5. Esta es primitiva recursiva en virtud del item 9 y del Ejercicio 5.

$\dashv$

Por ejemplo, podemos aplicar estos resultados para demostrar que la relación

$$\{< x, y > \mid (\exists q < y + 1)[x \cdot q = y]\}$$

es recursiva primitiva. Hacemos esto observando la forma en que está escrita la línea anterior y luego completando los detalles. En primer lugar, la relación ternaria

$$R_1(x, y, q) \Longleftrightarrow x \cdot q = y$$

se obtiene de la relación de igualdad sustituyendo las siguientes funciones $< x, y, q > \mapsto x \cdot q$ y I_2^3. En segundo lugar, una aplicación del teorema anterior muestra que la relación ternaria

$$R_2(x, y, z) \Longleftrightarrow (\exists q < z)[x \cdot q = y]$$

es recursiva primitiva. Finalmente aplicamos sustitución:

$$(\exists q < y + 1)[x \cdot q = y] \Longleftrightarrow R_2(x, y, y + 1)$$

En resumen, podemos demostrar que esta relación es recursiva primitiva examinando la forma sintáctica de su definición y verificando que se ha construido utilizando sólo piezas que se sabe que son recursivas primitivas.

14. La relación de divisibilidad $x|y$, es decir, la relación

$$\{< x, y > \mid x \text{ divide } y \text{ con resto } 0\},$$

es recursiva primitiva. (Aquí adoptamos la convención de que 0 se divide a sí mismo, pero no divide a ningún entero positivo). Esto se debe a que

$$\begin{aligned} x|y &\Longleftrightarrow \text{ para algún cociente } q, \text{ tenemos que } x \cdot q = y \\ &\Longleftrightarrow (\exists q \leqslant y)[x \cdot q = y] \\ &\Longleftrightarrow (\exists q < y + 1)[x \cdot q = y]. \end{aligned}$$

Es decir, ¡la relación que examinamos en el Ejemplo anterior no es más que la relación de divisibilidad!

En efecto, estamos construyendo un determinado lenguaje de tal manera que se garantiza que cualquier función o relación definible en el lenguaje será recursiva primitiva. (Para la divisibilidad, el hecho crucial fue que la expresión "$(\exists q < y + 1)[x \cdot q = y]$" pertenecía a este lenguaje). Este lenguaje incluye lo siguiente:

- Variables: Las funciones de proyección son recursivas primitivas.
- Constantes (numerales): Las funciones constantes son recursivas primitivas.
- Símbolos de función: Podemos usar símbolos para cualquier función recursiva primitiva en la lista que se está acumulando, de acuerdo a las convenciones usuales ($+, \times, \dot{-} \ldots$ y habrá más por venir).
- Combinaciones: $\sum_{x<y}, \prod_{x<y}$, y habrá más por venir.
- Símbolos de relación: Podemos usar símbolos para cualquier relación recursiva primitiva en la lista que estamos construyendo ($\leqslant, =, |, \ldots$, con más por venir).
- Más combinaciones: "no", "y", "o" se pueden aplicar a las relaciones.
- "Cuantificadores" acotados: $\forall x < y$ y $\exists x < y$. (Aquí se necesita el límite superior y).

Tenemos teoremas que nos aseguran que las funciones o relaciones expresables en este lenguaje seguramente serán recursivas primitivas.

Por ejemplo, a continuación agregamos el conjunto de números primos (como una relación unaria) a nuestra lista:

15. El conjunto $\{2, 3, 5, \ldots\}$ de números naturales primos (como relación unaria sobre $\mathbb{N}$) es recursivo primitivo. Para ver esto, observe que

$$x \text{ es primo } \iff 1 < x \text{ y } (\forall u < x)(\forall v < x)[uv \neq x],$$

y el lado derecho está escrito en el lenguaje disponible para nosotros.

2.1.1. Búsqueda acotada

El operador de búsqueda (a menudo llamado minimización u operador μ) proporciona una forma útil de definir una función en términos de una "búsqueda" para la primera vez que se cumple una condición dada.

Definición 2.1.1 Para una relación P de aridad $(k+1)$, el número $(\mu t < y)P(\vec{x}, t)$ se define mediante la ecuación:

$$(\mu t < y)P(\vec{x}, t) = \begin{cases} \text{el mínimo } t \text{ tal que } t < y \text{ y } P(\vec{x}, t), \text{ si existe} \\ y \text{ si no existe tal } t \end{cases}$$

Por ejemplo, si definimos

$$f(x,y) = \mu t < y[t \text{ es primo y } x < t]$$

entonces $f(6,4) = 4$ y $f(6,8) = f(6,800) = 7$.

Teorema 2.1.1 Si P es una relación primitiva recursiva, entonces la función

$$f(\vec{x},y) = (\mu t < y)P(\vec{x},t)$$

es una función recursiva primitiva.

Prueba: Aplicaremos la recursividad primitiva. Trivialmente $f(\vec{x},0) = 0$, por lo que aquí no hay problema. El problema es ver cómo $f(\vec{x},y+1)$ (llámelo b) depende de $f(\vec{x},y)$ (llámelo a):

- Si $a < y$, entonces $b = a$. (La búsqueda para números menores que y tuvo éxito).
- Si $a = y$ y $P(\vec{x},y)$, entonces $b = y$.
- En caso contrario, $b = y+1$.

Así, si definimos

$$g(a,\vec{x},y) = \begin{cases} a & \text{si } a < y \\ y & \text{si } a \not< y \text{ y } P(\vec{x},y) \\ y+1 & \text{si } a \not< y \text{ y no } P(\vec{x},y), \end{cases}$$

entonces f se obtiene por recursión primitiva de las funciones $\vec{x} \mapsto 0$ y g. Como P es una relación recursiva primitiva, se deduce que g es recursiva primitiva (por definición, por casos) y, por tanto, f es recursiva primitiva. ⊣Hay otra prueba de este teorema, que se basa en la siguiente notable ecuación:

$$(\mu t < y)P(\vec{x},t) = \sum_{u<y} \prod_{t \leqslant u} C_{\overline{P}}(\vec{x},t)$$

Un operador de búsqueda relacionado es una *maximización* acotada. Defina el operador $\overline{\mu}$ de la siguiente manera:

$$(\overline{\mu} t < y)P(\vec{x},t) = \begin{cases} \text{el número } t \text{ más grande tal que } t \leqslant y \text{ y } P(\vec{x},t), \text{ si lo hay} \\ 0 \quad \text{si no existe tal } t \end{cases}$$

Teorema 2.1.2 Si P es una relación recursiva primitiva, entonces la función

$$f(\vec{x}, y) = (\overline{\mu}t < y)P(\vec{x}, t)$$

es una función recursiva primitiva.
Prueba:

$$(\overline{\mu}t < y)P(\vec{x}, t) = y \dot{-} (\mu s < y)P(\vec{x}, y \dot{-} s).$$

Esta ecuación capta la idea de buscar hacia abajo a partir de y. $\dashv$

Euclides observó que el conjunto de los números primos es ilimitado. Por tanto, la función

$$h(x) = \text{el menor número primo mayor que } x$$

es total. También es primitiva recursiva porque

$$h(x) = \mu t < (x! + 2)[t \text{ es primo y } x < t].$$

El límite superior $x! + 2$ es suficiente porque para cualquier factor primo p de $x! + 1$, tenemos $x < p \leqslant x! + 1$. Así que cualquier búsqueda de un número primo mayor que x no necesita ir más allá de $x! + 1$.

Digresión: Aquí hay un resultado interesante en la teoría de números. El "postulado de Bertrand" establece que para cualquier $x > 3$, siempre habrá un número primo p con $x < p < 2x - 2$. (El postulado de Bertrand implica que en el párrafo anterior basta con usar simplemente $h(x) = \mu t < (2x - 2)[t \text{ es primo y } x < t]$.) En 1845, el matemático francés Joseph Bertrand, usando tablas de números primos, verificó esta afirmación para x por debajo de tres millones.

Luego, en 1850, el ruso P.L. Chebyshev (Tchebychef) demostró el resultado en general. En 1932, el húngaro Paul Erdős dio una prueba mejor, que ahora se puede encontrar en los libros de texto universitarios de teoría de números. El origen de

Chebyshev lo dijo
Así que lo diré de nuevo
Siempre hay un primo
Entre N and $2N$

se desconoce, lo cual puede ser mejor.

Defina p_x como el $(x+1)$ primer número primo, de modo que

$$p_0 = 2, p_1 = 3, p_2 = 5, p_3 = 7, p_4 = 11,$$

etcétera. En otras palabras, p_x es el x-ésimo primo impar, excepto que $p_0 = 2$. (El teorema de los números primos nos dice que p_x crece a una velocidad similar a $x \ln x$, pero eso no viene al caso).

16. La función $x \mapsto p_x$ es recursiva primitiva porque tenemos las ecuaciones de recursión

$$\begin{aligned} p_0 &= 2 \\ p_{x+1} &= h(p_x), \end{aligned}$$

donde h es la función anterior que encuentra el siguiente primo.

Es fácil ver que siempre tenemos $x + 1 < p_x$; una prueba formal puede utilizar la inducción.

Necesitaremos métodos para codificar una cadena de números mediante un solo número. Un método que es conceptualmente simple utiliza potencias de números primos. Definimos la "notación entre corchetes" de la siguiente manera.

$$\begin{aligned} [] &= 1 \\ [x] &= 2^{x+1} \\ [x, y] &= 2^{x+1}3^{y+1} \\ [x, y, z] &= 2^{x+1}3^{y+1}5^{z+1} \\ &\cdots \\ [x_0, x_1, \ldots, x_k] &= 2^{x_0+1}3^{x_1+1} \cdots p_k^{x_k+1}. \end{aligned}$$

Por ejemplo, $[2, 1] = 72$ y $[2, 1, 0] = 360$. Claramente, para cualquier valor de k, la función

$$< x_0, x_1, \ldots, x_k > \mapsto [x_0, x_1, \ldots, x_k]$$

es una función recursiva primitiva de aridad $(k + 1)$. Pero codificar es inútil, a menos que podamos decodificarlo. (El "teorema fundamental de la aritmética" es la afirmación de que todo número entero positivo tiene una factorización en números primos, única excepto por el orden.

Para decodificar, estamos explotando implícitamente la unicidad de la factorización de números primos). El ítem 17 dará una función de decodificación recursiva primitiva.

Digresión: usar potencias de números primos no es de ninguna manera la única forma de codificar una cadena de números. Es un método muy conveniente para nuestros propósitos actuales, pero existen otros métodos. Aquí hay un enfoque muy diferente:

$$< x_0, x_1, \ldots, x_k > \mapsto 1\underbrace{00\cdots0}_{x_0}1\underbrace{00\cdots0}_{x_1}1\cdots\cdots1\underbrace{00\cdots0}_{x_k}$$

Es decir, una secuencia de longitud n puede codificarse mediante el número cuya representación binaria tiene n unos. El número de ceros que siguen al i-ésimo 1 en la representación corresponde al i-ésimo componente de la secuencia.

Aquí hay unos ejemplos:

$$\begin{aligned} < 0, 3, 2 > &\mapsto 11000100 = 196 \\ < 2, 1, 0 > &\mapsto 100101 = 37 \\ <> &\mapsto 0 = 0 \\ < 7 > &\mapsto 10000000 = 128 \\ < 0, 0, 0, 0 > &\mapsto 1111 = 15 \end{aligned}$$

Estos valores se pueden comparar con los valores arrojados por la notación entre corchetes: $[0, 3, 2] = 2^1 \cdot 3^4 \cdot 5^3 = 20,250$, $[2, 1, 0] = 2^3 \cdot 3^2 \cdot 5^1 = 360$, $[] = 1$, $[7] = 2^8 = 256$, $[0, 0, 0, 0] = 2 \cdot 3 \cdot 5 \cdot 7 = 210$.

En particular, supongamos que queremos codificar una secuencia de dos números. Este método produce

$$< m, n > \mapsto 1\underbrace{00\cdots0}_{m}1\underbrace{00\cdots0}_{n} = 2^{m+n+1} + 2^n = 2^n(2^{m+1} + 1).$$

La notación entre corchetes produce simplemente $[m, n] = 2^{m+1} \cdot 3^{n+1}$. Ambas "funciones de emparejamiento" tienen la característica de que crecen exponencialmente a medida que m y n aumentan.

Curiosamente, existen funciones de emparejamiento *polinomial*, y aquí hay una:

$$J(m, n) = \frac{1}{2}((m + n)^2 + 3m + n).$$

La función J es uno a uno, por lo que el par $< m, n >$ es recuperable a partir del valor $J(m, n)$. De hecho, la función J asigna $\mathbb{N} \times \mathbb{N}$ uno a uno a $\mathbb{N}$.

¿Y de dónde viene J? Aquí hay una pista. Calcule $J(m, n)$ para todos los valores pequeños de m y n, digamos $m + n \leqslant 4$. Luego haga una gráfica en el plano, colocando el número $J(m, n)$ en el punto del plano con coordenadas $< m, n >$. Compruebe si está surgiendo un patrón.

17. Existe una función primitiva recursiva de "decodificación" de aridad dos, cuyo valor en $< x, y >$ se escribe $(x)_y$, con la propiedad de que siempre que $y \leqslant k$,

$$([x_0, x_1, \ldots, x_k])_y = x_y.$$

Esto es

(código para una secuencia)$_y$ = el $(y + 1)$-ésimo término de la secuencia.

Por ejemplo, $(72)_0 = 2$ y $(72)_1 = 1$ porque $72 = [2, 1]$.

Primero, observe que el exponente de un primo q en la factorización de un entero positivo x es

$$\mu e \left(q^{e+1} \nmid x\right),$$

la e más pequeña para la cual $e + 1$ sería demasiado. Podemos acotar la búsqueda en x porque si $q^e \mid x$, entonces $e < q^e \leqslant x$. Es decir, el exponente de q en la factorización de x es

$$(\mu e < x) \left(q^{e+1} \nmid x\right).$$

Ahora supongamos que el primo q es p_y. Definimos

$$(x)_y^* = (\mu e < x) \left(p_y^{e+1} \nmid x\right)$$

así que $(x)_y^*$ es el exponente de p_y en la factorización prima de x.

En segundo lugar, para nuestra función de decodificación, necesitamos uno menos que el exponente del primo p_y en la factorización del código de secuencia. En consecuencia, definimos

$$(x)_y = (x)_y^* \dot{-} 1 = (\mu e < x) \left(p_y^{e+1} \nmid x\right) \dot{-} 1.$$

El lado derecho de esta ecuación está escrito en nuestro lenguaje, por lo que la función es recursiva primitiva. La función prueba potencias de p_y hasta que encuentra la más grande en la factorización de x, y luego retrocede en 1. Si p_y no divide a x, entonces $(x)_y = 0$, lo cual es bastante inofensivo. También $(0)_y = 0$, pero por una razón diferente.

18. Digamos que y es un *número de secuencia* si $y = []$ o $y = [x_0, x_1, \ldots, x_k]$ para algún k y algunos $x_0, x_1, \ldots, x_k$. Por ejemplo, 1 es un número de secuencia pero 50 no lo es.

    El conjunto de números de secuencia es recursivo primitivo; consulte el ejercicio 14. El conjunto de números de secuencia comienza como $\{1, 2, 4, 6, 8, 12, \ldots\}$.

19. Existe una función recursiva primitiva lh tal que

$$\mathrm{lh}[x_0, x_1, \ldots, x_k] = k + 1.$$

    Por ejemplo, $\mathrm{lh}(360) = 3$. Aquí "lh"significa "longitud". Definimos

$$\mathrm{lh}(x) = (\mu k < x)\,(p_k \nmid x)\,.$$

    Así, por ejemplo, $\mathrm{lh}(50) = 1$.

    De su definición se desprende que esta función es recursiva primitiva. El límite superior de la búsqueda μ es adecuado porque si $p_{k-1} \mid x$, entonces $(k-1)+1 < p_{k-1} \leqslant x$.

    Si s es un número de secuencia de longitud positiva, entonces

$$(s)_{\mathrm{lh}(s) \dot{-} 1}$$

    será el último componente de la secuencia.

20. Existe una función recursiva primitiva de aridad dos cuyo valor en $< x, y >$ se llama *restricción* de x a y, escrita $x \upharpoonright y$, con la propiedad de que siempre que $y \leqslant k + 1$ entonces

$$[x_0, x_1, \ldots, x_k] \upharpoonright y = [x_0, x_1, \ldots, x_{y-1}].$$

    Es decir, la restricción de x a y nos da los primeros y componentes de la secuencia.

Definimos

$$x \upharpoonright y = \prod_{i<y} p_i^{(x)_i^*}$$

Por ejemplo, si s es un número de secuencia, entonces $s \upharpoonright (\mathrm{lh}(s) \dot{-} 1)$ codificará el resultado de eliminar el último elemento de la secuencia, si lo hubiera.

21. Existe una función recursiva primitiva de aridad dos cuyo valor en $< x, y >$ se llama *concatenación* de x y y, escrita $x * y$, con la propiedad de que siempre que x y y sean números de secuencia, entonces $x * y$ es el número de secuencia de longitud $\mathrm{lh}(x) + \mathrm{lh}(y)$ cuyos componentes son primero los componentes de x y luego los componentes de y.

    Definimos

$$x * y = x \cdot \prod_{i<\mathrm{lh}(y)} p_{i+\mathrm{lh}(x)}^{(y)_i^*}.$$

    Por ejemplo, $72 * 72 = [2, 1, 2, 1] = 441,000$. Si s es un número de secuencia, entonces $s * [x]$ codificará el resultado de unir x al *final* de la secuencia.

22. También podemos definir una operación "asterisco mayúsculo". Sea

$$*_{t<y} a_t = a_0 * a_1 * \cdots * a_{y-1}$$

    (agrupados a la izquierda). Si f es una función recursiva primitiva de aridad $(k+1)$, entonces también lo es la función cuyo valor en $< \vec{x}, y >$ es $*_{t<y} f(\vec{x}, t)$, como se puede ver en el par de ecuaciones recursivas:

$$\begin{aligned} *_{t<0} f(\vec{x}, t) &= 1 \\ *_{t<y+1} f(\vec{x}, t) &= *_{t<y} f(\vec{x}, t) * f(\vec{x}, y) \end{aligned}$$

    Para cualquier función f de aridad $(k+1)$, definimos $\bar{f}$ mediante la ecuación

$$\bar{f}\,(\vec{x}, y) = [f(\vec{x}, 0), f(\vec{x}, 1), \ldots, f(\vec{x}, y-1)]$$

    de modo que el número $f(\vec{x}, y)$ codifica y valores de f, es decir, los valores $f(\vec{x}, t)$ para todo $t < y$. Por ejemplo, $f(\vec{x}, 0) = [] = 1$, codifica 0 valores. Y $f(\vec{x}, 2) = [f(\vec{x}, 0), f(\vec{x}, 1)]$. Claramente, $f(\vec{x}, y)$ es siempre un número de secuencia de longitud y.

23. Si f es recursiva primitiva, entonces $\bar{f}$ también lo es porque

$$\bar{f}\ (\vec{x}, y) = \prod_{i<y} p_i^{f(\vec{x},i)+1}.$$

Ahora supongamos que tenemos una función g de aridad $(k+2)$. Entonces existe una función única f de aridad $(k+1)$ que satisface la ecuación

$$f(\vec{x}, y) = g(\bar{f}\ (\vec{x}, y), \vec{x}, y)$$

para todo $\vec{x}$ y y. Por ejemplo,

$$\begin{aligned} f(\vec{x}, 0) &= g([], \vec{x}, 0) = g(1, \vec{x}, 0) = g(1, \vec{x}, 0) \\ f(\vec{x}, 1) &= g([f(\vec{x}, 0)], \vec{x}, 1) \\ f(\vec{x}, 2) &= g([f(\vec{x}, 0), f(\vec{x}, 1)], \vec{x}, 2) \end{aligned}$$

etcétera. La función f se determina de forma recursiva; podemos encontrar $f(\vec{x}, y)$ después de conocer $f(\vec{x}, t)$ para todo $t < y$.

24. Supongamos que g es una función recursiva primitiva de aridad $(k+2)$, y sea f la única función de aridad $(k+1)$ para la cual

$$f(\vec{x}, y) = g(\bar{f}\ (\vec{x}, y), \vec{x}, y)$$

para todo $\vec{x}$ y y. Entonces f también es recursiva primitiva.

Para ver que f es recursiva primitiva, primero examinamos $\bar{f}$. tenemos el par de ecuaciones de recursión

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

de lo cual vemos que $\bar{f}$ es recursiva primitiva. En segundo lugar, la recursividad primitiva de f se deriva de la ecuación

$$f(\vec{x}, y) = g(\bar{f}\ (\vec{x}, y), \vec{x}, y)$$

una vez que sabemos que $\bar{f}$ es recursiva primitiva.

La definición de recursividad primitiva implicó definir el valor de una función en términos de su valor inmediatamente anterior. El ítem 24 muestra que obtenemos una ventaja adicional: el valor de una función se puede definir en términos de todos sus valores anteriores.

Hasta ahora hemos visto que muchas de las funciones cotidianas de los números naturales son recursivas primitivas. Pero la clase de funciones recursivas primitivas no incluye todas las funciones sobre $\mathbb{N}$ que uno consideraría efectivamente calculables.

W. Ackermann mostró cómo construir una función efectivamente calculable que crezca más rápido que cualquier función recursiva primitiva. Además, podemos "diagonalizar" las funciones recursivas primitivas. A grandes rasgos, así es como funcionaría: cualquier función recursiva primitiva está determinada por un árbol, lo que muestra cómo se construye a partir de funcioes iniciales usando composición y recursión primitiva.

Podemos, con algo de esfuerzo, codificar estos árboles mediante números naturales. La función "universal"

$$\Psi(x,y) = \begin{cases} f(x) & \text{si } y \text{ codifica un árbol para una función recursiva} \\ & \text{primitiva de aridad uno} \\ 0 & \text{en otro caso} \end{cases}$$

es efectivamente calculable (y total). Pero $\Psi(x,x)+1$ y $1 \dot{-} \Psi(x,x)$ son funciones totales efectivamente calculables que no pueden ser recursivas primitivas.

2.2. Operación de búsqueda

Obtenemos la clase de funciones parciales recursivas generales al permitir que las funciones se construyan mediante el uso de la búsqueda (además de la composición y la recursividad primitiva). La búsqueda (también llamada minimización) corresponde a un operador-μ ilimitado. Para una función parcial

g de aridad $(k+1)$, definimos

$$\mu y[g(\vec{x},y)=0] = \begin{cases} \text{el menor número } y \text{ tal que } g(\vec{x},y)=0 \text{ y para todo } t \\ \text{menor que } y, \text{ el valor } g(\vec{x},t) \text{ está definido y es distinto} \\ \text{de cero, si existe tal } y \\ \text{indefinido, si no existe tal } y. \end{cases}$$

Esta cantidad puede no estar definida para algunos (o todos) los valores de $\vec{x}$, incluso si g resulta ser una función total.

Ejemplo: Supongamos que conocemos la siguiente información sobre la función g:

$$\begin{array}{ll} g(0,0)=7 & g(0,1)=0 \\ g(1,0)=\uparrow & g(1,1)=0 \end{array}$$

Entonces $\mu y[g(0,y)=0]$ es 1 y $\mu y[g(1,y)=0]$ está indefinida.

Se dice que una función parcial h de aridad k se obtiene a partir de g mediante *búsqueda* si la ecuación

$$h(\vec{x}) = \mu y[g(\vec{x},y)=0]$$

se cumple para todo $\vec{x}$, con el entendimiento habitual de que para que una ecuación se cumpla, ambos lados no están definidos o ambos lados están definidos y son iguales.

Luego decimos que una función parcial es *recursiva general* si se puede construir a partir de las funciones cero, sucesor y proyección, donde se nos permite usar composición, recursividad primitiva y búsqueda.

La colección de funciones parciales recursivas generales incluye todas las funciones recursivas primitivas (que son todas totales) y más. Como ejemplo extremo, la función vacía de aridad uno (es decir, la función con dominio vacío) es una función parcial recursiva general; se obtiene mediante búsqueda a partir de la función constante $g(x,y)=3$.

9A. Si f es una función parcial recursiva general, entonces también lo son las funciones s y p:

$$s(\vec{x},y) = \sum_{t<y} f(\vec{x},t) \quad \text{y} \quad p(\vec{x},y) = \prod_{t<y} f(\vec{x},t).$$

Para cualquier $\vec{x}$ particular, estas funciones se definen para todo y o para un segmento inicial finito de los números naturales.

Definimos una relación R como una *relación recursiva general* si su función característica C_R (que por definición es siempre total) es una función recursiva general. Como caso especial de búsqueda, siempre que R es una relación recursiva general de aridad $(k+1)$, entonces la función h de aridad k definida por la ecuación

$$h(\vec{x}) = \mu y R(\vec{x}, y)$$

es una función parcial recursiva general.

Nuevamente tenemos una regla de sustitución: siempre que Q sea una relación recursiva general de aridad n y $g_1, \ldots, g_n$ sean funciones recursivas generales totales de aridad k, entonces la relación de aridad k

$$\{\vec{x} \mid < g_1(\vec{x}), \ldots, g_n(\vec{x}) > \in Q\}$$

es recursiva general porque su función característica se obtiene de C_Q y $g_1, \ldots, g_n$ por composición. Pero esto no es necesariamente cierto si las funciones g_i no son totales. En ese caso, la composición no nos da C_Q completo, sino una subfunción no total del mismo.

Por ejemplo, para cualquier función recursiva general total f, su gráfica

$$\{\vec{x}, y \mid f(\vec{x}) = y\}$$

es una relación recursiva general. (De manera similar, la gráfica de cualquier función recursiva primitiva será una relación recursiva primitiva). Veremos más adelante que esto puede fallar en el caso de una función no total.
Teorema:

(d) Si Q y R son relaciones recursivas generales de aridad k, entonces también lo son $\overline{Q}$, $Q \cap R$ y $Q \cup R$.

(e) Si Q es una relación recursiva general de aridad $(k+1)$, entonces también lo son las relaciones

$$\{< \vec{x}, y > \mid (\forall t < y) Q(\vec{x}, t)\} \quad \text{y} \quad \{< \vec{x}, y > \mid (\exists t < y) Q(\vec{x}, t)\}.$$

La prueba no ha cambiado.

La definición por casos sigue siendo válida, pero debemos ser más cuidadosos con su demostración. Supongamos que g es una función parcial recursiva

general de aridad k y que Q es una relación recursiva general de aridad k. Defina g^Q por la ecuación

$$g^Q(\vec{x}) = \begin{cases} g(\vec{x}) & \text{si } Q(\vec{x}) \\ 0 & \text{si no } Q(\vec{x}) \end{cases}$$

Entonces g^Q también es una función parcial recursiva general. Pero no podemos escribir simplemente $g^Q(\vec{x}) = g(\vec{x}) \cdot C_Q(\vec{x})$ porque puede haber algunas $\vec{x}$ que no estén en el dominio de g (por lo que el lado derecho no estará definido) y no en Q (por lo que el lado izquierdo será 0). En cambio, primero podemos usar la recursividad primitiva para construir la función.

$$G(\vec{x}, 0) = 0$$
$$G(\vec{x}, y+1) = g(\vec{x})$$

que es como g excepto que tiene un "interruptor de encendido y apagado". Entonces tenemos la ecuación

$$g^Q(\vec{x}) = G(\vec{x}, C_Q(\vec{x})).$$

mostrando que g^Q es una función parcial recursiva general.

Ahora, si también tenemos otra función parcial recursiva general f de aridad k y definimos

$$h(\vec{x}) = \begin{cases} f(\vec{x}) & \text{si } Q(\vec{x}) \\ g(\vec{x}) & \text{si no } Q(\vec{x}) \end{cases}$$

entonces h es una función parcial recursiva general porque $h(x) = f^Q(\vec{x}) + g^{\overline{Q}}(\vec{x})$.

24A. Asuma que g es una función parcial recursiva general de aridad $(k+2)$ y sea f la única función de aridad $(k+1)$ para la cual

$$f(\vec{x}, y) = g(\overline{f}(\vec{x}, y), \vec{x}, y)$$

para todo $\vec{x}$ y y. (Si g no es total, entonces es posible que para algunos valores de $\vec{x}$, la cantidad $f(\vec{x}, y)$ se defina sólo para un número finito de ys.) Entonces f también es una función parcial recursiva general.

La prueba es como antes.

Anteriormente se argumentó que la colección de funciones recursivas primitivas no puede contener todas las funciones totales efectivamente calculables. Pero la tesis de Church implica que la colección de funciones parciales recursivas generales las contiene todas, así como las funciones no totales efectivamente calculables. Como se ha indicado informalmente, no es posible "diagonalizar" la colección de funciones parciales recursivas generales.

2.2.1. Ejercicios

1. ¿Entiendes la recursividad primitiva? ¿Eres positivo? Si eres positivo, ve al Ejercicio 2.

2. Resta 1. Ve al ejercicio 1.

3. Dé un árbol de construcción completo para la multiplicación (item 3).

4. Demuestre que la función que eleva al cuadrado $f(x) = x^2$ es recursiva primitiva proporcionando un árbol de construcción que muestre en detalle cómo se puede construir a partir de funciones iniciales mediante el uso de composición y recursividad primitiva. (En las hojas del árbol, debes tener sólo funciones iniciales; por ejemplo, si quieres usar la suma, debes construirla).

5. Demuestre que la función

$$\text{pos}(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x = 0 \end{cases}$$

   es recursiva primitiva dando un árbol de construcción.

6. Demuestre que la función de paridad

$$C_{\text{odd}}(x) = \begin{cases} 1 & \text{si } x \text{ es impar} \\ 0 & \text{si } x \text{ es par} \end{cases}$$

   es recursiva primitiva dando un árbol de construcción.

7. Demuestre que la función $< x, y > \mapsto |x - y|$ es recursiva primitiva.

8. Demuestre que la función $< x, y > \mapsto \text{máx} < x, y >$ es recursiva primitiva.

9. Demuestre que la función $< x, y > \mapsto$ mín $< x, y >$ es recursiva primitiva.

10. Utilice el postulado de Bertrand para demostrar (por inducción) que $p_x \leqslant 2^{x+1}$ y que la igualdad se cumple sólo para $x = 0$.

Capítulo 3

Programas y máquinas

En este capítulo nos centramos en otra forma de formalizar el concepto de calculabilidad efectiva: los programas de máquinas de registros. Nuestro primer objetivo es mostrar que todas las funciones parciales recursivas generales también son computables mediante máquinas de registros. Este hecho nos permitirá aplicar nuestro trabajo en el Capítulo 2 para ver que muchas funciones cotidianas son computables por máquinas de registros. Usando esto, podremos construir un programa universal, es decir, un programa para calcular la función parcial $\Phi(w, x)$ = el resultado de aplicar el programa codificado por w a la entrada x.

3.1. Máquinas de registros

Recuerde del Capítulo 1 que un programa de máquina de registros es una secuencia finita de instrucciones de los siguientes tipos:

- "Incrementa r", $I\ r$ (donde $0 \leqslant r \leqslant K$): El efecto de esta instrucción es aumentar el contenido del registro r en 1. Luego, la máquina pasa a la siguiente instrucción del programa (si corresponde).

- "Decrementa r", $D\ r$ (donde $0 \leqslant r \leqslant K$): El efecto de esta instrucción depende del contenido del registro r. Si ese número es distinto de cero, se decrementa en 1 y la máquina no pasa a la siguiente instrucción, sino a la siguiente de la siguiente. Pero si el número en el registro r es cero, la máquina simplemente pasa a la siguiente instrucción. En resumen,

la máquina intenta disminuir el registro r y, si tiene éxito, se salta una instrucción.

- "Jump q", $J\ q$ (donde q es un número entero: positivo, negativo o cero): todos los registros se dejan sin cambios. La máquina toma como siguiente instrucción la $q-$ésima instrucción que sigue a ésta en el programa (si $q \geqslant 0$), o la $|q|-$ésima instrucción que precede a ésta (si $q < 0$). La máquina se detiene si no existe tal instrucción en el programa. La instrucción $J\ 0$ da como resultado un bucle, en el que la máquina ejecuta esta instrucción una y otra vez.

Ahora supongamos que f es una función parcial de aridad n sobre $\mathbb{N}$. Posiblemente habrá un programa $\mathcal{P}$ tal que si iniciamos una máquina de registros (que tiene todos los registros a los que $\mathcal{P}$ se refiere) con $x_1, \ldots, x_n$ en los registros $1, \ldots, n$ y 0 en los demás registros, y aplicamos el programa $\mathcal{P}$, entonces se cumplen las siguientes condiciones:

- Si $f(x_1, \ldots, x_n)$ está definida, entonces el cálculo finalmente termina con $f(x_1, \ldots, x_n)$ en el registro 0. Además, el cálculo termina buscando la $(p+1)-$ésima instrucción, donde p es la longitud de $\mathcal{P}$.

- Si $f(x_1, \ldots, x_n)$ no está definida, entonces el cálculo nunca termina.

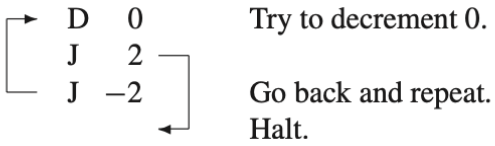
Si existe tal programa $\mathcal{P}$, decimos que $\mathcal{P}$ calcula f. (Observe que cuando iniciamos la ejecución de un programa, hay tres posibilidades: (I) podría ejecutarse para siempre; (II) podría detenerse "bien", al buscar la primera instrucción que no está allí; (III) podría detenerse "mal", ya sea intentando saltar a algún lugar antes del inicio del programa o intentando avanzar a una instrucción, inexistente, después de la última instrucción del programa. En nuestra definición de "$\mathcal{P}$ calcula f", hemos elegido descartar paradas incorrectas. Esto será conveniente para ejecutar programas de principio a fin).

Por ejemplo, la función de suma $x + y$ se calcula mediante un programa de máquina de registros dado en el Capítulo 1. Además, en el Capítulo 1, vimos algunas subrutinas básicas:

CLEAR r	Borra el registro r.
MOVE from r to s	Se borra el registro r.
COPY from r to s using t	El registro r no cambia.

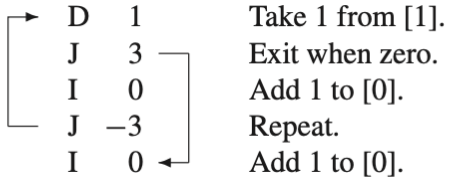
Estas subrutinas tienen longitud 3, 7 y 15, respectivamente.

Para un ejemplo trivial, la función constante cero de aridad n se calcula mediante el programa vacío y por nuestro programa de tres líneas para borrar el registro 0:



(Las flechas son para ayudarnos a ver lo que hace el programa). Además, al agregar algunos instrucciones de incremento, podemos ver que cualquier función constante total se calcula mediante algún programa de registros.

La función sucesora de aridad uno $S(x) = x + 1$ es calculada por el programa que mueve el contenido del registro 1 (llame a este número [1]) al registro 0 y luego incrementa el número:



La función de proyección I_n^k es calculada por el programa de siete líneas que mueve el contenido del registro n al registro 0.

A continuación, queremos mostrar que la clase de funciones parciales calculadas por programas de máquinas de registros está cerrada bajo composición. Es decir, queremos saber que siempre que tengamos programas de máquinas de registros que calculen $f, g_1, \ldots, g_n$ y

$$h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_n(\vec{x})),$$

entonces podemos producir un programa para h. Esto implica encadenar varios programas. Pero se debe tener cuidado para asegurarse de que un programa no tropiece con la basura dejada por un programa anterior y que no borre los datos necesarios para un programa posterior.

Sabemos lo que significa que $\mathcal{P}$ calcule f: cuando le proporcionamos a $\mathcal{P}$ las condiciones ideales (la entrada en los registros $1, \ldots, k$, los otros registros

están vacíos), entonces $\mathcal{P}$ regresará (en el registro 0) el valor de la función, si está definido.

Pero necesitamos un programa que sea menos complicado. ¿Qué pasa si las condiciones no son ideales? Supongamos que la entrada está en los registros $r_1, \ldots, r_k$, donde estos son k números distintos (no necesariamente consecutivos y no necesariamente en orden creciente). Y supongamos que no queremos prometer que los demás registros estén vacíos. Además, queremos un programa que no borra ni altera el contenido de los primeros s registros, para algún número grande s queremos que la información de esos registros se mantenga segura.

Esto es lo que queremos, formulado como una definición:
Definición: Supongamos que f es una función parcial de aridad k, $\mathcal{Q}$ es un programa, $r_1, \ldots, r_k$ son números naturales distintos, y s y t son números naturales. Entonces decimos que $\mathcal{Q}$ calcula f a partir de los registros $r_1, \ldots, r_k$ en t preservando s si cada vez que iniciamos una máquina de registros (con suficientes registros) con el programa $\mathcal{Q}$ y con $a_1, \ldots, a_k$ en los registros $r_1, \ldots, r_k$ entonces no importa lo que haya inicialmente en los otros registros, tenemos los siguientes resultados finales:

- Si $f(a_1, \ldots, a_k)$ está definida, entonces el cálculo finalmente se detiene (es decir, se detiene buscando la primera instrucción inexistente, la $(q+1)$-ésima instrucción, donde q es la longitud de $\mathcal{Q}$), con el valor $f(a_1, \ldots, a_k)$ en t. Además, los primeros s registros, los registros $0, 1, \ldots, s-1$, contienen los mismos números que tenían inicialmente, excepto posiblemente el de t.

- Si $f(a_1, \ldots, a_k)$ no está definida, entonces el cálculo nunca se detiene.

Como caso especial, podemos decir que si $\mathcal{Q}$ calcula f a partir de $1, \ldots, k$ en 0 preservando 0, entonces $\mathcal{Q}$ calcula f (en el sentido definido originalmente). Lo contrario no es del todo cierto debido a la cuestión de si los registros están inicialmente vacíos.

El siguiente lema dice que podemos tener lo que pide la definición anterior.
Lema: Supongamos que el programa $\mathcal{P}$ calcula la función parcial f de aridad k. Sean $r_1, \ldots, r_k$ números naturales distintos. Sean t y s números naturales. Entonces podemos encontrar un programa $\mathcal{Q}$ que calcule f a partir de $r_1, \ldots, r_k$ en t preservando s.
Prueba: Sea M la *dirección* más grande en $\mathcal{P}$ (es decir, el número más grande tal que alguna instrucción de incremento o decremento en $\mathcal{P}$ direcciona el

registro M). Probablemente $M > k$; si no luego aumente M para que sea $k + 1$. Sea j el mayor de los números $s, r_1, \ldots, r_k$. Aquí está el programa $\mathcal{Q}$:

COPY	r_1 to $j + 1$ usando $j + 2$
COPY	r_2 to $j + 2$ usando $j + 3$
$\ldots$	
COPY	r_k to $j + k$ usando $j + k + 1$
CLEAR	j
CLEAR	$j + k + 1$
CLEAR	$j + k + 2$
$\ldots$	
CLEAR	$j + M$
$\mathcal{P}$	reubicado por j
MOVE	de j a t (si $j \neq t$)

Aquí, "reubicado por j" significa que j se agrega a la dirección de todas las instrucciones de incremento y decremento. Por tanto, el programa reubicado opera en los registros $j, j + 1, \ldots, j + M$ exactamente como $\mathcal{P}$ operó en los registros $0, 1, \ldots, M$. Dado que $\mathcal{P}$ calcula f, el programa reubicado dejará $f(a_1, \ldots, a_k)$, si está definido, en el registro j. El programa $\mathcal{Q}$ entonces mueve j al registro t. Exceptuando al registro t, el programa deja los registros $0, 1, \ldots, j - 1$ sin alterar. El siguiente mapa ilustra cómo $\mathcal{Q}$ usa los registros:

registro	0	intacto
registro	1	intacto
	$\vdots$	
registro	$j - 1$	intacto
registro	j	salida
registro	$j + 1$	entrada
	$\vdots$	
registro	$j + k$	entrada
registro	$j + k + 1$	espacio de trabajo
	$\vdots$	
registro	$j + M$	espacio de trabajo

El lema será una herramienta útil siempre que necesitemos unir diferentes programas. Como nuestra primera aplicación de este lema, podemos demostrar que la clase de funciones parciales computables por máquinas de registros es cerrada bajo composición.

Teorema: La clase de funciones parciales computables por máquinas de registros es cerrada bajo composición. Es decir, siempre que tengamos programas de máquinas de registros que calculen funciones parciales $f, g_1, \ldots, g_n$ de las cuales se obtiene h por composición, entonces podemos hacer un programa que calcula la función parcial h.

Prueba: Supongamos que la función parcial h de aridad k se obtiene por composición de f y $g_1, \ldots, g_n$:

$$h(\vec{x}) = f(g_1(\vec{x}), \ldots, g_n(\vec{x})).$$

Supongamos además que tenemos programas que calculan $f, g_1, \ldots, g_n$. queremos hacer un programa que calcule h. El cual es el siguiente:

Calcule g_1 a partir de $1, \ldots, k$ a $k+1$, conservando $k+1$.
Calcule g_2 a partir de $1, \ldots, k$ a $k+2$, conservando $k+2$.
$\ldots$
Calcule g_n a partir de $1, \ldots, k$ a $k+n$, conservando $k+n$.
Calcule f a partir de $k+1, \ldots, k+n$ a 0, conservando 0.

Aquí, nos basamos en el lema para proporcionar los componentes del programa. Observe que para que el programa se detenga, necesitamos que las funciones $g_1(\vec{x}), \ldots, g_n(\vec{x})$ estén definidas y necesitamos que f esté definida en $g_1(\vec{x}), \ldots, g_n(\vec{x})$. $\dashv$

Por ejemplo, supongamos que h está dada por la ecuación $h(x, y, z) = f(g(z, y), 7, x)$ y tenemos programas de registros para f y g. Del teorema anterior se deduce que h es computable por alguna máquina de registros. Podemos escribir

$$h(x, y, z) = f(g(I_3^3(x, y, z), I_2^3(x, y, z)), K_7(x, y, z), I_1^3(x, y, z))$$

(donde K_7 es una función constante) y aplicar el teorema dos veces, primero para obtener $g(I_3^3(x, y, z), I_2^3(x, y, z))$ y luego para obtener h. La moraleja de

este ejemplo es que podemos poner libremente las variables donde queramos y aplicar composición con funciones de proyección para justificar lo que hemos hecho. En particular, si

$$h(x_1, x_2, \ldots, x_m) = f(\_\_\_, \_\_\_, \ldots, \_\_\_),$$

donde cada espacio en blanco se llena con algún x_i o alguna constante, entonces a partir de un programa para f podemos obtener un programa para h.

Aún así, este capítulo aún no ha producido un programa para una sola función "interesante". Para eso, necesitamos un resultado de cerradura más: cerradura bajo recursividad primitiva. Es decir, queremos saber que si h se obtiene de f y g mediante recursividad primitiva

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

y tenemos programas de registros para f y g, entonces podemos obtener un programa para h. O en el caso de que $\vec{x}$ esté vacío

$$h(0) = m$$
$$h(y+1) = g(h(y), y)$$

(para algún número m) y tenemos un programa para g, entonces queremos saber si podemos obtener un programa para h.

De ello se deducirá que todas las funciones recursivas primitivas (en particular, las del Capítulo 2) son computables por máquinas de registros. Y finalmente, veremos que la clase de funciones computables de máquinas de registros incluye mucho más que los ejemplos simplistas con los que comenzamos.

Teorema: La clase de funciones parciales computables por máquinas de registros es cerrada bajo recursividad primitiva.

Prueba: Supongamos que h es la función parcial de aridad $(n+1)$ obtenida por recursión primitiva a partir de las funciones parciales f y g:

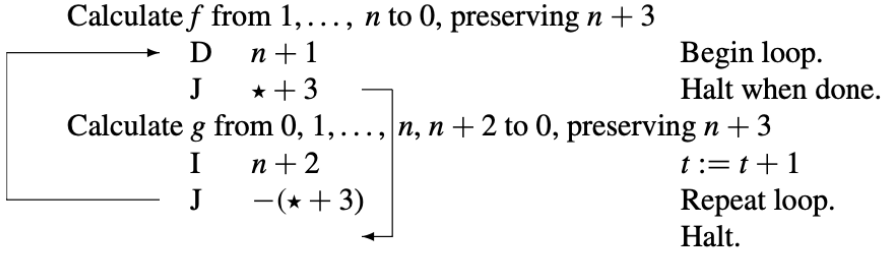
$$h(\vec{x}, 0) = f(\vec{x})$$
$$h(\vec{x}, y+1) = g(h(\vec{x}, y), \vec{x}, y)$$

Supongamos que tenemos programas de registros para f y g. Necesitamos hacer un programa para h.

El programa comenzará con $x_1, \ldots, x_n, y$ en los registros $1, 2, \ldots, n, n+1$. El programa pondrá $h(\vec{x}, t)$ en el registro 0 primero para $t = 0$, luego para $t = 1$, y así sucesivamente hasta $t = y$. El número t se mantiene en el registro $n + 2$, que inicialmente contiene 0. El siguiente mapa ilustra este uso:

registro	0	$h(\vec{x}, t)$
registro	1	x_1
	$\vdots$	
registro	n	x_n
registro	$n+1$	$y - t$
registro	$n+2$	t
registro	$n+3$	espacio de trabajo
	$\vdots$	

Aquí está el programa:



Aquí $\star$ es la longitud del programa que se utiliza para g.

Para ver la corrección de este programa, establecemos lo siguiente:

Afirmación: Cada vez que llegamos a la instrucción D $n + 1$ (al comienzo del ciclo), después de ejecutar el ciclo k veces,

- el registro 0 contiene $h(\vec{x}, k)$,
- el registro $n + 1$ contiene $y - k$,
- el registro $n + 2$ contiene k.

La afirmación de que existen "invariantes de ciclo" se demuestra por inducción (como es habitual en programas con ciclos) sobre k.

Para $k = 0$, cuando lleguemos a D $n + 1$ sin haber ejecutado el bucle en absoluto, el registro 0 contiene $f(\vec{x})$, que es $h(x, 0)$, el registro $n + 1$ no se modifica, por lo que todavía contiene y, y el registro $n + 2$ sigue siendo 0.

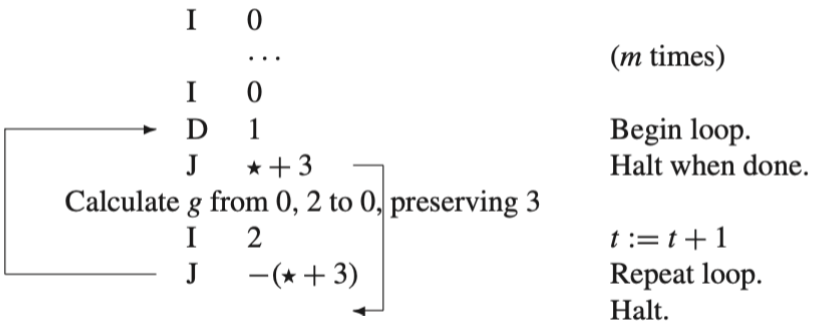
Ahora vamos al paso inductivo. En el $(k + 1)$-ésimo paso por el ciclo, decrementamos el registro $n + 1$ (según la hipótesis inductiva, anteriormente contenía $y - k$, por lo que ahora es $y - (k + 1)$), incrementamos el registro $n + 2$ (anteriormente contenía k, por lo que ahora es $(k + 1)$, y en el registro 0 ponemos $g(h(\vec{x}, k), \vec{x}, k)$, que de hecho es $h(\vec{x}, k + 1)$.

Entonces, por inducción, la afirmación se cumple cada vez que iniciamos el ciclo. El programa se detiene cuando iniciamos el ciclo con 0 en el registro $n + 1$. En este punto, hemos hecho el ciclo y veces (por la afirmación), y el registro 0 contiene $h(\vec{x}, y)$, como se deseaba.

El programa se modifica fácilmente para el caso en que $\vec{x}$ sea vacío. Queremos usar los registros de la siguiente manera:

registro	0	$h(t)$
registro	1	$y - t$
registro	2	t
registro	3	espacio de trabajo
	$\vdots$	

Aquí está el programa:



El argumento de la corrección sigue siendo aplicable con $n = 0$. ⊣

Reuniendo los resultados hasta el momento, llegamos a la siguiente conclusión.

Teorema: Toda función recursiva primitiva es computable por máquinas de registros.

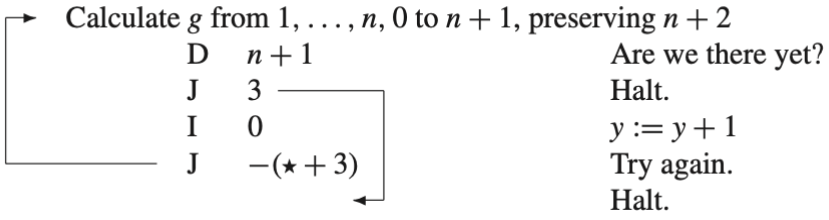
Este teorema promete programas de máquinas de registros para todas las funciones recursivas primitivas del capítulo anterior. Pero, por supuesto, podemos hacerlo mejor:

Teorema: Toda función parcial recursiva general es computable por máquinas de registros.

Prueba: Necesitamos agregar el operador-μ

$$h(\vec{x}) = \mu y[g(\vec{x}, y) = 0].$$

Usamos el programa obvio:



El programa utiliza los registros de la siguiente manera:

registro	0	y
registro	1	x_1
	$\vdots$	
registro	n	x_n
registro	$n+1$	$g(\vec{x}, y)$
registro	$n+2$	espacio de trabajo
	$\vdots$	

Por supuesto, es posible que este programa nunca se detenga. ⊣

Este teorema da la mitad de un hecho significativo, formalizar el concepto de calculabilidad efectiva mediante recursividad general y formalizar el concepto de calculabilidad efectiva mediante máquinas de registros conduce a exactamente la misma clase de funciones parciales. Pronto llegaremos a la otra mitad. En particular, el teorema ilustra que las máquinas de registros son capaces de hacer mucho más de lo que podría, inicialmente, a partir de su muy simple definición.

3.2. Un programa Universal

En el Capítulo 1, se argumentó que la función parcial "universal"

$$\Phi(e, x) = \begin{array}{l}\text{el resultado de aplicar el programa} \\ \text{codificado mediante } e \text{ a la entrada } x\end{array}$$

debería ser una función parcial computable. Ahora planeamos verificar este hecho en el caso de programas de máquinas de registros.