Damiano Zanardini

Teoría de la Computabilidad

De los resultados clásicos al día a día de la Informática

Damiano Zanardini

Teoría de la Computabilidad

De los resultados clásicos al día a día de la Informática

ISBN : 978-84-941071-7-7
Editorial : Administraciondigital S.L
Printed in Spain – Impreso en España
Madrid 2015

Índice

Capítulo 1

Introducción

Supongamos que Alice y Bob[1] están implementando un sistema informático. Alice se encuentra con un fragmento de código, P, que quiere reutilizar, pero no está segura de lo que el código hace. Por esto pide a Bob que le diga lo que P calcula[2]:

```
1  proc p(a:int) returns int {
2    return a*2;
3  }
4
5  var x:int, y:int;
6  x := 20;
7  y := p(x)
```

Es muy fácil entender el resultado del cálculo de P, así que Bob puede contestar a la pregunta en pocos segundos y sin ninguna dificultad.

Ahora supongamos que Alice encuentra otro fragmento de código, Q, y, otra vez, quiere usarlo en su sistema. Para ello, pide a Bob que eche un

[1] La elección de estos dos nombres no es nada original, ya que se usan muy a menudo en este tipo de ejemplos, sobre todo en física y criptografía, cuando hace falta hablar de dos agentes que comunican entre ellos; son nombres cómodos porque concuerdan con las dos primeras letras del alfabeto, así que "Alice dice a Bob" es una manera de decir "la persona A dice a la persona B".

[2] Este programa está escrito en el lenguaje descrito en el capítulo 5; se trata de una lenguaje muy sencillo que cualquier informático con conocimientos de algún lenguaje imperativo puede entender fácilmente.

vistazo a Q y le diga lo que calcula[3]:

```
1  proc p(a:int) returns int {
2    var c1:int, c2:int, c3:int;
3    c1 := a / 3;
4    c2 := a % 3 - 1;
5    b := c2;
6    c3 := c1;
7    while (c3>0) do {
8      b := b + c3 % 3 + 2;
9      c3 := c3 - 1; }
10   if (c2 > 1) then {
11     b := b + 2
12   } else {
13     b := b + c2 + 1; }
14   b := b + c1 * 3;
15   if (c1 % 3 = 2) then {
16     b := b + c3;
17   } else { b := b + 1; }
18   return b;
19 }
20
21 var x:int, y:int;
22 x := 20;
23 y := p(x)
```

Es evidente que entender lo que hace Q es mucho más difícil, así que Bob tarda un buen rato en contestar. Sin embargo, el resultado final es que... ¡P y Q calculan exactamente lo mismo!

En general, Bob podrá "descifrar" (un informático podría decir "decompilar") cualquier fragmento de código, y encontrará una respuesta para todas las preguntas que Alice le pueda poner. Lo malo es que podría tardar muchísimo (no sería muy difícil producir programas mucho más largos y más complejos que sigan calculando la misma función). Por esto, los informáticos intentan desarrollar unos programas, que llamaremos *analizadores*, que realicen el trabajo de Bob tardando mucho menos y con un mayor nivel de fiabilidad.

Lo que pasa es que *ni siquiera el analizador más potente podrá contestar a todas las preguntas de Alice*: siempre habrá un programa para el que no sabrá dar una respuesta (en este caso, que el valor final de `y` es 40).

En este curso vamos a entender por qué pasa esto.

[3]Para los que tengan dudas, `/` es la división entera y `%` es el resto.

1.1 De qué va este libro

Este libro estudia los límites de la computación.

Las preguntas que intenta contestar son (1) teóricas: ¿Qué es la computación? ¿Qué se puede y qué no se puede calcular? ¿Cómo afecta un cambio de formalismo o de lenguaje a nuestra capacidad de calcular? (2) prácticas: ¿Qué implicaciones tienen los resultados de computabilidad en el día a día de un informático? ¿Puedo construir un analizador que me diga lo que quiero saber de una clase de programas (es decir, si tienen o no ciertas propiedades)? ¿cómo? ¿con qué dificultades y limitaciones me voy a enfrentar? (3) e incluso filosóficas: ¿Qué implicaciones tienen estos resultados en nuestra idea del mundo? y muchas más.

Contestar la primera clase de preguntas necesita una definición de la idea de *cálculo*. No fue fácil encontrar tal definición. De hecho, el gran lógico Kurt Gödel tardó años en convencerse de que la definición propuesta por Alan Turing era plausible [7].

Sólo contestando la primera pregunta, podemos esperar encontrar una buena respuesta para las demás, sobre todo *¿qué se puede y qué no se puede calcular?* y *¿cómo afecta un cambio de formalismo o de lenguaje a nuestra capacidad de calcular?*. La respuesta que tenemos desde hace 70 años no es un teorema sino una "tesis", la *Tesis de Church*, es decir, algo de que no tenemos una demostración formal. Sin embargo, mucha observación y reflexión, además de una serie de teoremas de equivalencia entre diversos formalismos de computación (incluyendo algunos que, en un principio, parecían destinados justamente a demostrar su falsedad), llevaron los informáticos[4] a asumir la Tesis de Church como algo *verdadero*. Hay que decir también que la Tesis de Church está constantemente bajo el ataque de científicos que pretenden demostrar su falsedad inventando formalismos de computación que supuestamente no la respetan. En todo caso, el consenso general acepta la tesis, aunque la partida no está del todo cerrada y nuevos descubrimientos en el campo de la biología o de la física podrían representar serias amenazas para ella.

Este libro, a pesar de su enfoque marcadamente teórico, pretende también considerar la segunda clase de preguntas (las preguntas por así decir "prácticas") y dar algunas respuestas. Áreas de la informática como el *Análisis Estático* encuentran a diario sus límites en los resultados de la Teoría de la Computabilidad, por lo que podemos decir que los resultados de un puñado de matemáticos y lógicos en los años 30 del siglo XX siguen mermando la posi-

[4]En ese momento de la historia se consideraban, y eran, matemáticos; la informática como tal todavía no existía.

bilidad de comprobar de forma exacta muchas propiedades importantes de los programas, tan importantes que hasta la seguridad de transacciones bancarias o medios de transporte depende de ellas.

Finalmente, los límites de la computación tienen también implicaciones en el estudio del hombre, especialmente en el ámbito de la *Inteligencia Artificial*, concretándose en lo que se suele llamar el *problema mente-máquina*. ¿Es el hombre, por muy complejo que sea, una máquina? ¿O es algo más, cuyas capacidades las máquinas nunca podrán alcanzar? ¿Cuándo se podrá simular *en todo* un ser humano por medio de un robot, si es que algún día se podrá? Estos temas siguen abiertos y el debate entre las varias escuelas de pensamiento se desarrolla tanto en el ámbito ciéntifico como filosófico.

1.2 Un vistazo al temario

Después de introducir algunas ideas fundamentales como la de algoritmo, vamos a describir (Parte I) algunos formalismos que pretenden expresar la idea de *computación*: *Máquinas de Turing*, lenguajes *For* y *While*, y *Funciones Recursivas*. Ciertos resultados de equivalencia entre algunos de estos formalismos nos llevan a la *Tesis de Church*, que nos acompañará durante todo este curso.

Siguiendo las geniales ideas de Gödel, vamos a definir el proceso de *aritmetización* de los conceptos de teoría de la Computabilidad, que, junto con otros teoremas y resultados, nos permitirá demostrar que ciertos problemas (o lenguajes) son o no *recursivos* o *recursivamente enumerables* (Parte II). Esta discusión, que se desarrolla en el ámbito de la Informática, está estrechamente relacionada con dos teoremas de Lógica demostrados por el mismo Gödel: los *teoremas de incompletitud*. De hecho, estos teoremas son anteriores a la Tesis de Church, por lo que será interesante ver las implicaciones de la incompletitud en la idea de computabilidad.

Este libro nace, entre otras cosas, de la certeza que la Teoría de la Computabilidad ne es únicamente algo teórico, sin implicaciones en lo que se llama "vida real" (sin ir más allá, nos referimos, al menos, a la vida real de un informático o de un científico). La última parte del libro (Parte III) intentará introducir los temas del *Análisis Estático* y (más superficialmente) el *Problema Mente-Máquina* desde la perspectiva de la Teoría de la Computabilidad.

1.3 Presentación de la bibliografía

Nuestra discusión parte de los trabajos de grandes lógicos y matemáticos de la década del 1930: Kurt Gödel [9], Alan M. Turing [20, 21], y Alonzo Church [3]. En estos destacados artículos se encuentran las bases de la investigación de la teoría de la computabilidad. Aunque a menudo el formalismo en el que se expresan las ideas en estos artículos no es tan pulido como el que se ha ido afirmando a lo largo de los años, puede que no sea demasiado atrevido afirmar que contienen, por lo menos *in nuce*, mucho del material que tanto daría de pensar a científicos y filósofos hasta nuestros días. Aunque difícil, su lectura es bastante recomendable, y quizá resulta menos ardua si se aprovecha ediciones como las de Martin Davis [7] y Manuel Garrido [9], o libros explicativos como el de Ernest Nagel, James R. Newman y Jean-Yves Girard [13] (que incluyen al mismo Gödel entre los autores).

Entre los textos clásicos de Teoría de la Computabilidad, nos apoyamos principalmente en Harry R. Lewis y Christos H. Papadimitriou [12], sin olvidar otros compendios bien conocidos como Nigel J. Cutland [6], S. Barry Cooper [4], Jr. Harley Rogers [18], George S. Boolos and Richard C. Jeffrey [2] y Piergiorgio Odifreddi [15]. Entre los tratados que, aún sin poder definirse tratados de Teoría de Computabilidad, proporcionan ideas y reflexiones interesantes sobre el tema, destacamos un artículo de Herbert A. Simon [19] y el ambicioso libro de Douglas R. Hofstadter [10] que, a partir del título, intenta llevar a la luz relaciones ocultas entre lógica, dibujo y música.

Finalmente, acerca de las dos "extensiones" de la Teoría de la Computabilidad que este curso pretende tocar, proponemos el libro clásico de informática de John Hopcroft, Rajeev Motwani y Jeffrey Ullman [11] y el trabajo de Flemming Nielson, Hanne R. Nielson y Chris Hankin [14], que se ocupa más específicamente de análisis estático. Por lo que tiene que ver con el problema mente-máquina, tal vez el trabajo de Roger Penrose [16] puede ser un buen punto de partida, aunque la discusión está más que abierta y la "guerra de ideas" sigue generando trabajos y debates.

1.4 Ejercicios

Ejercicio 1.1 (Para reflexionar) *Intuitivamente, ¿cómo se puede demostrar que el segundo programa siempre calcula el doble de su input? ¿Cuántos casos posible habría que considerar?*

Ejercicio 1.2 (Para reflexionar) *Pensar en ejemplos y casos concretos donde*

la posibilidad de escribir programas muy difíciles o imposibles de analizar (manteniendo su comportamiento) es una ventaja, bien desde el punto de vista de quien los escribe, bien de quien los va a ejecutar.

Capítulo 2

Algoritmos

La idea de *algoritmo* es algo que, en cuanto informáticos, conocemos al menos aproximadamente. Sin embargo, se trata de un concepto un tanto resbaladizo si lo que se pretende es investigar sus límites: por ejemplo, preguntarse si todos los formalismos de computación actuales están incluidos en las definiciones más clásicas de algoritmo, o la relación entre un algoritmo y la actividad de la mente humana. Para aclarar las ideas, necesitamos definir intuitivamente un *formalismo de computación*, o simplemente *formalismo*, como la descripción de un sistema en el que se puede llevar a cabo unos cálculos. Por ejemplo, todo lenguaje de programación es un formalismo de computación, al igual que una Máquina de Turing o una Función Recursiva (conceptos definidos más adelante) lo es.

Hay muchas definiciones de algoritmo (cuyo nombre se remonta probablemente al nombre del matemático persa Al Juarismi); todas intentan tener en cuenta eficazmente ciertas características que se creen estar incluidas en la idea misma de *computación*. A lo largo de este curso, a partir del próximo capítulo, hablaremos muy a menudo de Alan M. Turing, inventor de las famosas *máquinas* que llevan su nombre. Probablemente, al definir estas máquinas, la idea de algoritmo que Turing tenía en la cabeza era algo como: un algoritmo es una serie de *instrucciones* con las siguientes propiedades:

- el número de instrucciones es *finito*, es decir, un algoritmo se puede escribir en su totalidad en una porción finita de espacio, ya sea una hoja de papel o un disco duro;
- cada instrucción tiene un efecto *limitado*, es decir, no puede modificar el

estado de algo que está fuera de su alcance;

- la computación se desarrolla en pasos *individuales* (una instrucción a la vez) y *discretos*;
- los pasos son *deterministas*: dependen únicamente de un número finito de pasos previos y de una cantidad finita de *datos*;
- no hay límite al número de pasos ni a la cantidad de *memoria* necesaria para almacenar los datos (finitos) iniciales, intermedios y finales.

Es fácil darse cuenta de que, hoy en día, existen muchas formulaciones del concepto de computación que no satisfacen del todo esta definición: de hecho, tenemos formalismos que (1) permiten ejecutar más de una instrucción a la vez y en varios sitios (*algoritmos paralelos*, hasta llegar a los *algoritmos distribuidos*); (2) cuyos pasos no son deterministas (*algoritmos randomizados* o *no deterministas*); (3) como ciertos agentes de *malware*, son programas (secuencias de instrucciones) que se automodifican; (4) disponen de una cantidad limitada de recursos (en realidad, este es el caso de muchas computadoras tradicionales que tienes una cantidad finita de memoria); o (5) al menos en teoría, no son discretos. Finalmente, nuevos paradigmas de computación como las *redes neuronales*, los *algoritmos genéticos* y la *computación cuántica* representan ulteriores desafíos a la idea clásica de algoritmo.

Afortunadamente, para lo que nos interesa, es muy fácil reconocer un algoritmo cuando lo vemos. Por lo tanto, de momento nos conformamos con la idea intuitiva de que un algoritmo es un procedimiento expresado en cierto formalismo que permite llevar a cabo unas computaciones.

2.1 Un poco de historia

Esta sección no pretende ser un relato histórico completo. Simplemente, queremos introducir algunas de las personas y de las ideas que pueblan los comienzos de la informática, desde los griegos hasta por lo menos la primera mitad del siglo XX.

Uno de los primeros ejemplos de algoritmo fue propuesto por Euclides para calcular el MCD de dos números. Se trata de una secuencia finita de instrucciones que permitía llevar a cabo el cálculo en un número finito de pasos. La idea de una *máquina* que pudiese calcular (ejecutar algoritmos) mecánicamente la tuvieron varios científicos a lo largo de la historia. En lugar de interesarse por cálculos matemáticos, el filósofo, matemático, jurista y político Gottfried

Wilhelm Leibniz (1646–1716) fue más original, y soñó con una máquina que diera un valor de verdad a *fórmulas lógicas*.

Este sueño siguió siendo tal hasta los comienzos del siglo XX, cuando David Hilbert, que entonces se encontraba en la cumbre de la fama como matemático, propuso en Paris en 1900 sus famosos *23 problemas* sin solución que, en su opinión y en la realidad, iban a dirigir la investigación matemática durante todo el siglo[1]. Entre ellos, destaca el décimo problema: encontrar un *algoritmo* para determinar si una *ecuación diofántica*[2] polinomial con coeficientes enteros tiene una solución entera.

28 años después, el mismo Hilbert propone el *problema de la decisión*, también conocido como *Entscheidungsproblem*, que tiene relación con el sueño de Leibniz y el décimo problema: encontrar un algoritmo general que decida[3] si una fórmula del *cálculo de primer orden*[4] es un teorema.

Todo esto nos lleva a la puertas de 1930. Muchos resultados de esos años se hallaron en al ambiente privilegiado de Princeton, donde trabajaban algunas de las mejores mentes del planeta: "*Princeton in the 1930's was an exciting place for logic. There was Church together with his students Rosser and Kleene. There was John von Neumann. Alan Turing who had been thinking about the notion of effective calculability, came as a visiting graduate student in 1936 and stayed to complete his Ph. D. under Church. And Kurt Gödel visited the Institute for advanced Study in 1933 and 1935, before moving there permanently.*" [8]. En Princeton, los matemáticos buscaban una buena definición de "lo que es calculable", o "lo que es efectivo". En 1936, Alonzo Church y Alan M. Turing llegaron de manera independiente a sus propuestas [3, 20], que se afirmaron, respectivamente, como la tesis de Church y la definición de la Máquina de Turing.

Estos resultados demostraron que el *Entscheidungsproblem* tiene solución negativa (es decir, el algoritmo buscado no existe), y todas las formas entonces conocidas de computación (propuestas por Church, Kleene, Rosser, Herbrand, Gödel) son equivalentes a las Máquinas de Turing. Sin embargo, pasaron años para que Kurt Gödel y la comunidad científica aceptasen las Máquinas de Turing como el formalismo "privilegiado" para expresar la idea de computación en la Teoría de la Computabilidad.

---

[1] Y también durante el siglo XXI, ya que algunos de estos problemas siguen sin solución.

[2] Las ecuaciones diofánticas son ecuaciones polinomiales como $x^2 + y^2 = z^2$, planteadas sobre el conjunto de los enteros o naturales.

[3] El concepto de *decisión* se detallará más adelante (Sec. 3.3); de momento decidir es contestar correctamente que sí o que no a una pregunta.

[4] Se trata de la *Lógica de Primer Orden*, o *de los predicados*, que bien conocemos.

Parte I

Formalismos de computación

Capítulo 3

Máquinas de Turing

Este capítulo introduce el formalismo de computación "por excelencia", el que se afirmó en la decada de 1930 como la "piedra del parangón" a la que tenían que referirse los demás formalismos, y fue aceptado después de muchos años por Gödel como una descripción matemática satisfactoria de lo que significa "calcular" [7]. En este capítulo veremos que la fuerza de estas máquinas reside en que mejorar su *hardware* de cualquier forma nunca llega a ensanchar la clase de computaciones que pueden llevar a cabo (aunque sí puede mejorar, y mucho, su eficiencia). La presentación de este capítulo se basa principalmente en el libro de Lewis y Papadimitriou [12], pero puede haber pequeñas discrepancias porque hemos intentado tratar el tema de la forma más sencilla posible, limitándonos a lo estrictamente indispensable.

3.1 Definición

La primera definición de Máquina de Turing se encuentra en el artículo del mismo Alan M. Turing del 1936 "On Computable Numbers, with an Application to the Entscheidungsproblem" [20]. En este densísimo artículo de unas 32 páginas más apéndice se introduce la idea de Máquina de Turing, la máquina universal, los números computables, y el problema de la parada. Finalmente, el material introducido se aplica al *Entscheidungsproblem* o *Problema de la Decisión*. Turing se apoya en la aritmetización de Gödel. Para los jóvenes investigadores que se desesperan al ver rechazado su artículo de investigación, y que se sienten humiliados por los comentarios de los *reviewers*, quizá pueda

ser un consuelo leer lo que entonces anotó el revisor de este artículo:

This is a bizarre paper. It begins by defining a computing device absolutely unlike anything I have seen, then proceeds to show–I haven't quite followed the needlessly complicated formalism–that there are numbers that it can't compute. As I see it, there are two alternatives that apply to any machine that will ever be built: Either these numbers are too big to be represented in the machine, in which case the conclusion is obvious, or they are not; in that case, a machine that can't compute them is simply broken! Any tabulating machine worth its rent can compute all the values in the range it represents, and any number computable by a function–that is, by applying the four operations a number of times–can be computed by any modern tabulating machine since these machines–unlike the one proposed here with its bizarre mechanism–have the four operations hardwired. It seems that the "improvement" proposed by Turing is not an improvement over current technology at all, and I strongly suspect the machine is too simple to be of any use. If the article is accepted, Turing should remember that the language of this journal is English and change the title accordingly.

Indudablemente, leer esta reseña más de setenta años después y a la luz de la importancia del trabajo de Turing para toda la Informática nos hace sonreir, pero también nos enseña cómo una revisión demasiado superficial puede destrozar los mejores trabajos, incluso, como en este caso, piedras miliares de la Ciencia (con mayúscula).

Intuitivamente, una Máquina de Turing es una *cinta* infinita (tiene un inicio, pero no un fin) de casillas iguales. Una *cabeza* se mueve sobre la cinta: en cada paso, se puede mover como mucho una casilla hacia la derecha o una hacia la izquierda. La cabeza puede leer, escribir o borrar símbolos en la casilla en la que se encuentra (no puede observar otras casillas ni actuar sobre ellas). La cabeza se encuentra en cada instante en un *estado* interno y, en cada paso, actúa según este mismo estado y el símbolo leído en la cinta. La *acción* elegida en cada paso consiste en escribir un símbolo nuevo en la cinta o en mover la cabeza, y se acompaña a un cambio del estado interno. Más formalmente, una Máquina de Turing (MT)[1] es una quíntupla $\langle K, \Sigma, s, H, \delta \rangle$ donde

- K es el conjunto finito de los *estados internos* de la cabeza;
- Σ es el conjunto finito de los *símbolos* que pueden aparecer en la cinta;
- $s \in K$ es el *estado inicial*;

[1] Existen muchas definiciones de MT que difieren en algunos detalles; para lo que nos interesa, son todas equivalentes.

- $H \subseteq K$ es el conjunto de los *estados finales*: normalmente incluye el estado h de *terminación normal* (es decir, el estado en el que la cabeza no tiene nada más que hacer y termina su trabajo) y el estado e de *error* (es decir, el estado en el que se encuentra la cabeza cuando algo ha ido mal);

- δ es la *función de transición*: su dominio es[2] $(K \setminus H) \times \Sigma$, es decir, el estado actual (que no puede ser un estado final) y el símbolo leído en la cinta; su codominio es $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$, es decir, δ devuelve el nuevo estado interno y una acción a ejecutar (escribir un símbolo en la cinta o mover la cabeza).

El significado de δ es el siguiente: su dominio representa el input (la condición inicial) en cada paso: la cabeza se encuentra en un estado interno $q \in K \setminus H$ (es decir, no final) y observa el símbolo $\sigma \in \Sigma$ en la casilla donde se encuentra. El resultado $(q', x) = \delta(q, \sigma)$ es la acción ejecutada, o *paso*: la cabeza pasa del estado q al estado q' y

- si $x \in \Sigma$ entonces escribe el símbolo x en la cinta, y no se mueve;

- si $x = \leftarrow$ entonces da un paso a la izquierda;

- si $x = \rightarrow$ entonces da un paso a la derecha.

El conjunto Σ incluye un símbolo $\triangleright$ que marca el inicio de la cinta. Toda Máquina de Turing, al encontrar este símbolo, se mueve inmediatamente hacia la derecha sin cambiar de estado[3]. Σ también incluye el símbolo "␣", que indica que la casilla está vacía. La Figura 3.1 muestra una MT durante su funcionamento: hay 5 estados internos y actualmente la máquina se encuentra en q_2; la cabeza se encuentra sobre la tercera casilla, que contiene el símbolo a; en la cinta hay 4 casillas no vacías además de la primera que contiene $\triangleright$.

Una *configuración* es una descripción instantanea de la computación en un momento dado: incluye (1) el contenido de la cinta *leída hasta el momento*; (2) la posición de la cabeza en la cinta; y (3) su estado interno. La condición "leída hasta el momento" es fundamental, porque la cinta es infinita y representar su contenido sería imposible. Sin embargo, es fácil convencerse de que, si inicialmente sólo una parte finita de la cinta está escrita (veremos que este requisito es más que razonable), en ningún momento encontraremos infinitos

[2] El símbolo $\setminus$ representa la diferencia de conjuntos.

[3] Aunque no suele indicarse explicitamente en su definición, toda función de transición incluye este comportamiento, es decir, $\delta(q, \triangleright) = (q, \rightarrow)$ para toda δ y todo estado interno q.

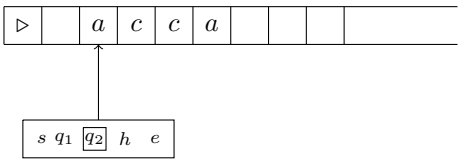


Figura 3.1: Una MT con su cinta (arriba) y su cabeza (con los estados internos)

símbolos en ella. Esto se debe a que, después de un número finito k de pasos de computación, la MT puede haber escrito símbolos como mucho en $k/2$ casillas[4], por lo que queda evidente que para escribir una cantidad infinita de símbolos se necesitaría una cantidad infinita de tiempo.

Esta observación nos permite representar una configuración con un número finito de símbolos: $(q, \triangleright\sigma_1..\sigma_n\underline{\sigma_{n+1}}\sigma_{n+2}..\sigma_m)$ donde q es el estado interno actual, y $\triangleright\sigma_1..\sigma_n\underline{\sigma_{n+1}}\sigma_{n+2}..\sigma_m$ significa que (1) el contenido de la porción de cinta leída hasta el momento es $\sigma_1..\sigma_m$; y (2) la cabeza se encuentra sobre el símbolo σ_{n+1}. La expresión $(q, \triangleright w\underline{\sigma}u) \vdash_M (q', \triangleright w'\underline{\sigma'}u')$ indica que la MT M pasa de la configuración $(q, \triangleright w\underline{\sigma}u)$ a $(q', \triangleright w'\underline{\sigma'}u')$ en un paso; $\vdash_M^k$ denota que lo hace en k pasos, y $\vdash_M^*$ denota que lo hace en 0 o más pasos[5].

En éste y otros ejemplos usaremos la *notación* unaria para representar los números naturales en la cinta, es decir, el alfabeto Σ es $\{a, \triangleright, \textvisiblespace\}$ por lo que sólo existe un símbolo además de $\triangleright$ y $\textvisiblespace$. El número n se representa simplemente con una serie de $n+1$ símbolos a seguidos. Representar n con $n+1$ en lugar de n símbolos evita que el número 0 "desaparezca", por así decir: para todo número natural tendremos al menos una a.

Ejemplo 3.1 (MT que escribe un número y termina) *La MT M_3 que escribe simplemente el número 3 y termina se define como:*

- *la configuración inicial es $(s, \triangleright\underline{\textvisiblespace})$, es decir, la cinta está vacía y la cabeza se encuentra en la segunda casilla (la primera después de $\triangleright$);*

- *la siguiente tabla representa δ; las primeras dos columnas representan el input y las otras dos especifican la acción a llevar a cabo:*

[4]En nuestra definición, la MT no puede escribir y moverse en un solo paso, por lo que deberá dedicar la mitad de los pasos a escribir y la otra mitad a moverse.

[5]En este contexto, el símbolo $*$ se llama estrella de Kleene, y significa "cero o más veces"; volveremos a verlo muchas veces en este curso.

estado	*símbolo*	*nuevo estado*	*acción*
s	$\sqcup$	q_0	$\rightarrow$
q_0	$\sqcup$	q_1	a
q_1	a	q_1'	$\rightarrow$
q_1'	$\sqcup$	q_2	a
q_2	a	q_2'	$\rightarrow$
q_2'	$\sqcup$	q_3	a
q_3	a	q_3'	$\rightarrow$
q_3'	$\sqcup$	q_4	a
q_4	a	q_4	$\leftarrow$
q_4	$\sqcup$	h	$\sqcup$
_	_	e	$\sqcup$

Lo que hace esta MT es

- *empezar con una cinta vacía, moverse hacia la derecha cuatro veces, y cada vez escribir a en la casilla correspondiente;*

- *cuando el número 3 aparece en la cinta en forma de aaaa, volver a la casilla inicial (permaneciendo en el estado q_4 hasta encontrar un $\sqcup$) y terminar la computación.*

La última línea de la tabla dice lo que ocurre cuando ninguna de las líneas anteriores es compatible con la configuración actual: se da un error y la MT termina en el estado de error. Esto puede pasar si, por ejemplo, la cinta no estaba inicialmente vacía. El símbolo _ significa "cualquiera" y se aplica tanto a estado internos como a símbolos.x

Este ejemplo es trivial porque la MT ni siquiera tiene que leer un input y siempre devuelve el mismo resultado (si necesitamos, por ejemplo, una máquina M_{11} que produzca el número 11, necesitamos definir otra MT con otra función δ). Sin embargo, nos permite empezar a entender cómo funciona una máquina y lo que esperamos que haga como regla general: escribir su resultado en la cinta y volver al principio de la misma cambiando el estado interno a h.

En general, se supone que el input de una MT empiece en la tercera casilla de la cinta, después de $\triangleright$ y de $\sqcup$, y que la cabeza se encuentre inicialmente en la segunda casilla. Normalmente, al final de la computación la cabeza volverá a encontrarse en la segunda casilla.

Ejemplo 3.2 (MT que suma dos números) *Esta MT M_+ que realiza una operación tan elemental como la suma necesita leer en la cinta la representación de los dos números n_1 y n_2 que tiene que sumar; al final de la computación el número $n_1 + n_2$ será el único contenido de la cinta. La configuración inicial es $(s, \triangleright\underline{\sqcup}a^{n_1+1}\sqcup a^{n_2+1})$ donde la expresión a^n representa una secuencia de n símbolos a seguidos (es decir, las dos secuencias son las representaciones de n_1 y n_2 en notación unaria). Lo que nos esperamos que haga esta máquina es (1) recorrer la cinta hasta encontrar el símbolo $\sqcup$ entre los dos números; (2) a partir de allí, "mover" la segunda secuencia de a una casilla hacia la izquierda, para que se fusione con la primera secuencia, y borrar la última a, dando el número $n_1 + n_2$; (3) volver al principio de la cinta y terminar. La siguiente tabla describe el comportamiento de esta MT; por comodidad, omitimos las operaciones triviales como volver al principio de la cinta (ya sabemos cómo se puede implementar), y los comportamientos erróneos. Mover la segunda secuencia de a una casilla hacia la izquierda se obtiene en realidad escribiendo una a al principio de la secuencia y borrando la última.*

estado	*símbolo*	*nuevo estado*	*acción*	
s	$\sqcup$	q_1	$\rightarrow$	
q_1	a	q_1	$\rightarrow$	*recorre* n_1
q_1	$\sqcup$	q_2	a	*llega al final de* n_1
q_2	a	q_2	$\rightarrow$	*recorre* n_2
q_2	$\sqcup$	q_3	$\leftarrow$	*llega al final de* n_2
q_3	a	q_4	$\sqcup$	*borra una a*
q_4	a	q_5	$\leftarrow$	
q_5	a	q_6	$\sqcup$	*borra otra a*
q_6	...	...	...	*vuelve y termina*

Notamos que esta máquina funciona incluso si al menos uno de los dos números es 0. Notamos también que el "cerebro" de esta máquina no necesita acordarse de cuántas casillas ha visto hasta el momento; este detalle es fundamental para que esta operación pueda ser implementada, ya que el número de estados internos es finito, por lo que la máquina "no tiene memoria suficiente" para tener un estado interno para cada número leído. Sin embargo, veremos que una MT sí tiene memoria suficiente para realizar todo tipo de operación: es la cinta, que es infinita (de hecho, n_1 y n_2 están guardados allí).

Ejemplo 3.3 (MT que termina sii el input es par) *Esta MT empieza en una configuración $(s, \triangleright\underline{\sqcup}a^{n+1})$ correspondiente al input n. La no terminación*

se obtiene con el estado q_3: si la máquina está en q_3 al terminar el número, entonces n es impar; por lo tanto, basta con que δ obligue la máquina a no terminar nunca (por ejemplo, moviendo la cabeza hacia la derecha infinitas veces, como en este ejemplo).

estado	*símbolo*	*nuevo estado*	*acción*	
s	$\sqcup$	q_1	$\rightarrow$	
q_1	a	q_2	$\rightarrow$	
q_2	a	q_3	$\rightarrow$	
q_2	$\sqcup$	q_4	$\leftarrow$	*el número es par*
q_3	a	q_2	$\rightarrow$	
q_3	$\sqcup$	q_3	$\rightarrow$	*el número es impar*
q_4	a	q_4	$\leftarrow$	
q_4	$\sqcup$	h	$\sqcup$	

Este último ejemplo nos dice que hay Máquinas de Turing que pueden realizar computaciones infinitas. Este hecho será muy importante en lo que sigue.

Hecho 3.1 *Existen MTs cuyas computaciones a veces no terminan (y también otras cuyas computaciones no terminan nunca, para ningún input).*

Los ejemplos vistos hasta ahora nos llevan a varias observaciones. A cada una de ellas sigue una breve discusión.

"Definir una MT, incluso sencilla, puede ser muy aburrido". En general es cierto. Por esto, la próxima sección introduce unas herramientas más cómodas para definir Máquinas de Turing *componiendo* piezas pequeñas en piezas más grandes.

"Las MTs son horriblemente lentas". Normalmente es cierto, si consideramos el número de pasos necesarios para realizar una tarea. Sin embargo, es muy importante subrayar que la Teoría de la Computabilidad *no* se preocupa por el *tiempo* que cierto formalismo tarda en hacer un trabajo, sino por si será capaz o no de hacerlo. La existencia de problemas indecidibles (Parte II) clarificará este asunto.

"Las cosas mejorarían, y mucho, cambiando ciertos detalles". Por ejemplo, es normal pensar que la notación unaria es muy pobre e ineficiente

con respecto a la notación decimal. Esto es cierto, si estamos hablando de eficiencia. Sin embargo, como acabamos de decir, una MT, aparte de tener una cinta infinita, también tiene una paciencia infinita, y la Teoría de la Computabilidad tampoco tiene mucha prisa. La Sección 3.4 dará una intuición de cómo ciertas mejorías "técnicas" no aumentan el poder computacional de una Máquina de Turing, aunque puedan hacerla mucho más eficiente.

3.2 Composición de Máquinas de Turing

Cuando hablamos de componer Máquinas de Turing no pensamos en distintas MTs, cada una con su cabeza, cinta y función de transición, que comunican entre ellas a través de algún sofisticado protocolo. Hablamos de fusionar distintas funciones de transición, cada una dedicada a realizar una parte de la computación principal en una única MT con una única cabeza y una única cinta.

Ejemplo 3.4 *Ya tenemos una MT M_+ que calcula la suma: podemos modificarla un poco para poder usarla como* componente *de la MT M_* que calcula el producto. Esta nueva M'_+ es casi igual que M_+, pero puede empezar su trabajo en cualquier configuración $(q, \triangleright w \underline{\sqcup} a^{n_1+1} \sqcup a^{n_2+1} \sqcup w')$, terminando en $(q', \triangleright w \underline{\sqcup} a^{n_1+n_2+1} \sqcup\sqcup w')$, para toda secuencia de símbolos w y w'. Es decir: M'_+ empieza a trabajar en un punto de la cinta que no tiene por qué ser la segunda casilla, y hace el trabajo de M_+* sin afectar *el resto de la cinta. q y q' son los estados internos en que M_* "deja el control" a M'_+ y lo retoma, respectivamente; durante su trabajo M'_+ pasará por cierto número de estados internos que funcionan como los de la M_+ original.*

Tenemos que definir algunas MTs básicas que hacen una parte pequeña de trabajo y paran, para poder componerlas formando máquinas más sofisticadas:

- R (resp., L) da un paso hacia la derecha (resp., izquierda) y termina: su función de transición es simplemente

estado	símbolo	nuevo estado	acción
s	$\_$	h	$\rightarrow$ (resp., $\leftarrow$)

que significa que, para cualquier estado incial y símbolo en la cinta, la acción y el estado final es el mismo: la MT mueve la cabeza y termina;

- R_σ (resp., L_σ) mueve hacia la derecha (resp., izquierda) hasta que encuentra el símbolo σ, y termina en esa misma casilla:

estado	símbolo	nuevo estado	acción
s	σ	h	σ
s	_	s	$\rightarrow$ (resp., $\leftarrow$)

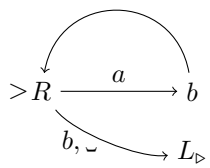
- $R_{\bar{\sigma}}$ (resp., $L_{\bar{\sigma}}$) mueve hacia la derecha (resp., izquierda) hasta que encuentra un símbolo que no sea σ, y termina en esa misma casilla:

estado	símbolo	nuevo estado	acción
s	σ	s	$\rightarrow$ (resp., $\leftarrow$)
s	_	h	σ

- M_σ, o simplemente σ, escribe el símbolo σ sin moverse:

estado	símbolo	nuevo estado	acción
s	_	h	σ

La composición de muchas MTs se representa con un diagrama como el que aparece aquí. El símbolo $>$ es el punto de entrada de la MT; M_1M_2 significa ejecutar M_1 e, inmediatamente después, M_2; una flecha etiquetada con un símbolo σ indica lo que pasa si el símbolo leído en la configuración actual es σ. Por ejemplo



significa (1) moverse una casilla a la derecha; (2) si el símbolo leído es a, entonces escribir una b (b es una forma abreviada de escribir M_b) y volver al principio; (3) si el símbolo leído es un ␣ o b, mover hacia el principio de la cinta (hasta encontrar $\triangleright$) y terminar. Una flecha etiquetada con varios símbolos (separados por una coma) indica que el control pasa de la primera a la segunda máquina si el símbolo leído es uno de los que aparecen en la etiqueta; una flecha sin etiqueta denota una transición incondicional, que no depende del símbolo

leído. Lo que hace esta MT es reemplazar todos los símbolos a al comienzo del input por b.

Queremos caracterizar la MT $M = \langle K, \Sigma, s, H, \delta\rangle$ dibujada arriba, que incluye R, M_b y $L_\triangleright$ como componentes. Sea $R = \langle K_R, \Sigma, s_R, H_R, \delta_R\rangle$, $M_b = \langle K_{M_b}, \Sigma, s_{M_b}, H_{M_b}, \delta_{M_b}\rangle$ y $L_\triangleright = \langle K_{L_\triangleright}, \Sigma, s_{L_\triangleright}, H_{L_\triangleright}, \delta_{L_\triangleright}\rangle$ (por comodidad, asumimos que el alfabeto es el mismo para las tres):

- $K = K_R \cup K_{M_b} \cup K_{L_\triangleright}$ (los tres conjuntos se suponen ser disyuntos);
- $s = s_R$ (porque la computación empieza por R);
- $H = H_{L_\triangleright}$ (porque la computación termina después de ejecutar $L_\triangleright$);
- para todo $\sigma \in \Sigma$ y $q \in K \setminus H$, la función $\delta(q, \sigma)$ se define como
  - $\delta(q, \sigma) = \delta_R(q, \sigma)$ si $q \in K_R \setminus H_R$;
  - $\delta(q, \sigma) = \delta_b(q, \sigma)$ si $q \in K_{M_b} \setminus H_{M_b}$;
  - $\delta(q, \sigma) = \delta_R(q, \sigma)$ si $q \in K_R \setminus H_R$;
  - $\delta(q, a) = (s_{M_b}, a)$ si $q \in H_R$ (si R ha acabado y la cabeza lee una a, el control pasa a M_b);
  - $\delta(q, b) = \delta(q, \textvisiblespace) = (\textvisiblespace, s_{L_\triangleright})$ si $q \in H_R$ (el control pasa a $L_\triangleright$;
  - $\delta(q, \sigma) = (s_R, \sigma)$ si $q \in H_{M_b}$ (al terminar M_b el control pasa a R).

Ejemplo 3.5 (Diagrama de M_+) *La MT M_+ descrita en el Ejemplo 3.2 se puede representar a través del siguiente diagrama*[6]*:*

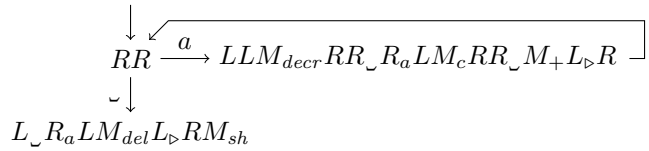
$$> RR_{\textvisiblespace} M_a R_{\textvisiblespace} L M_{\textvisiblespace} L M_{\textvisiblespace} L_\triangleright R$$

Ejemplo 3.6 (Producto) *La definición más básica del producto se basa en la suma: $m * 0 = 0$ y $m * (n + 1) = m + m * n$. Se trata básicamente de sumar n m veces, a partir de 0. La configuración inicial es $(s, \triangleright \underline{\textvisiblespace} a^{m+1} \textvisiblespace a^{n+1})$. Aparte de los que acabamos de definir, necesitamos otros ingredientes para definir M_*. Todas estas máquinas se pueden definir sin demasiados problemas; todas empiezan con la cabeza leyendo un $\textvisiblespace$, trabajan en la parte de cinta inmediatamente a la derecha de la cabeza, y terminan dejando la cabeza donde estaba.*

[6] En realidad hay pequeñas diferencias entre este diagrama y la tabla del Ejemplo 3.2, como el primer paso que en el diagrama ignora el símbolo leído, dando por descontado que es $\textvisiblespace$, mientras que la tabla parece indicar que la ausencia de $\textvisiblespace$ en la segunda casilla de la cinta generaría un error (que la tabla deja implícito por comodidad).

- *M_{decr} decrementa un número:* $(\_, \triangleright w \underline{\sqcup} a^{n+1}) \vdash^*_{M_{decr}} (\_, \triangleright w \underline{\sqcup} a^n)$ *(omitimos los estados porque no son importantes, y el resto de la cinta w');*
- *M_c copia un número n más allá de otro número p que se encuentra a su derecha en la cinta:* $(\_, \triangleright w \underline{\sqcup} a^{n+1} \sqcup a^{p+1}) \vdash^*_{M_c} (\_, \triangleright w \underline{\sqcup} a^{n+1} \sqcup a^{p+1} \sqcup a^{n+1})$*;*
- *M_{del} borra el primer número que encuentra, dejando una secuencia de $\sqcup$ en su lugar:* $(\_, \triangleright w \underline{\sqcup} \sqcup^k a^{n+1} u) \vdash^*_{M_{del}} (\_, \triangleright w \underline{\sqcup} \sqcup^{k+n+1} u)$*;*
- *M_{sh} mueve el número n hacia la izquierda eliminando todas las casillas vacías menos la primera:* $(\_, \triangleright \underline{\sqcup} \sqcup^* a^{n+1}) \vdash^*_{M_{sh}} (\_, \triangleright \underline{\sqcup} a^{n+1})$.

La MT M_* se puede representar con el siguiente diagrama:



La primera parte escribe el número 0 al final del input: la cabeza se mueve hacia la derecha hasta recorrer las representaciones de m y n ($R_a R_\sqcup R_a R_\sqcup R$), escribe una a que representa 0 y vuelve a la segunda casilla de la cinta ($L_\triangleright R$); los resultados parciales serán modificaciones de este último número, que llamaremos p. Después de esta operación, la cabeza compara m con 0: RR mueve la cabeza dos veces hacia la derecha, y si el símbolo leído es $\sqcup$ entonces $m = 0$ y se pasa a la parte final del cálculo; si no, la cabeza vuelve atrás (LL) y empieza el bucle. En cada iteración del bucle (1) M_{decr} decrementa m; (2) la cabeza se mueve hasta llegar a n ($RR_\sqcup R_a L$), y (3) a^{n+1} se copia después de p (M_c); (4) el número recién escrito se suma a p ($RR_\sqcup M_+$); y (5) la cabeza vuelve al principio ($L_\triangleright R$). La iteración se repite hasta que $m = 0$; en este caso se pasa a la parte final donde (1) se borra m ($L_\sqcup$, es decir, un paso a la izquierda y escritura de $\sqcup$); (2) se borra n de la cinta ($R_a LM_{del}$); (3) la cabeza vuelve al principio ($L_\triangleright R$); y (4) se mueve p hacia el principio de la lista. Al final p será $m * n$, y la configuración final será $(h, \triangleright \underline{\sqcup} a_{n*m+1})$.

3.3 Primeros elementos de computabilidad

Una de las preguntas más importantes en la Teoría de la Computabilidad es: ¿es el formalismo de computación A más o menos "expresivo" que el formalismo B? Para comparar distintos formalismos de computación es necesario

definir precisamente lo que significa *calcular*. Normalmente, calcular consiste en *decidir o semidecidir un lenguaje* o bien *calcular una función*.

3.3.1 Decidir un lenguaje

Se define un *lenguaje* sobre un alfabeto Σ_0 como un conjunto de palabras sobre este mismo alfabeto, es decir, un conjunto $L \subseteq \Sigma_0^*$ de cadenas $w \in \Sigma_0^*$, donde $*$ es la estrella de Kleene.

Ejemplo 3.7 *Las palabras correctas de un idioma son un ejemplo de lenguaje finito. El conjunto de los programas Java sintácticamente correctos es un lenguaje infinito. Dado* $\Sigma_0 = \{a, b\}$*, el conjunto de las palabras* $a^n b^n$ *con* $n \geq 0$ *es un lenguaje infinito, así como lo es el conjunto de las palabras (secuencias) palíndromas. Dado el mismo* $\Sigma_0 = \{a, b\}$*, el conjunto de las palabras* $a^{n+1}ba^{n^2+1}$ *para* $n \geq 0$ *(es decir, b es el símbolo que separa*[7] *dos números* m' *y* m'' *representados en notación unaria con* $m'' = m' * m'$*) es otro lenguaje infinito.*

Para hablar de MTs que deciden lenguajes necesitamos dos estados finales y y n ("yes" y "no"). Se dice que una MT *acepta* una cadena $w \in \Sigma_0^*$ si una computación con input w termina en el estado y, y que la *rechaza* si termina en el estado n. Estamos definendo estos conceptos para las MTs, pero valen para todo formalismo de computación, con algunas diferencias. El alfabeto Σ de la Máquina de Turing incluye Σ_0 más los símbolos $\triangleright$ y $\textvisiblespace$, y posiblemente otros que sean útiles para el cálculo.

Una MT M *decide* un lenguaje $L \subseteq \Sigma_0^*$ si para todo input $w \in \Sigma_0^*$:

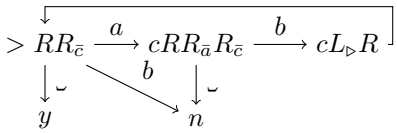
- si $w \in L$, entonces M acepta w;
- si $w \notin L$, entonces M rechaza w.

Un lenguaje se dice *decidible* o *recursivo*[8] o si existe una MT que lo decide.

Ejemplo 3.8 *El conjunto* $\{a^n b^n | n \geq 0\}$ *mencionado en el ejemplo anterior es decidido por la siguiente MT:*

[7] En M_+ y M_*, el símbolo separador es el mismo $\textvisiblespace$; sin embargo, al tener que decidir un lenguaje por parte de una MT, se prefiere que el input no contenga $\textvisiblespace$.

[8] Más adelante veremos que esta palabra se usa también para hablar de funciones definidas por recursión, al estilo de muchos languajes de programación; estos dos usos de la palabra están relacionados, pero hay que tener cuidado a la hora de usarlos.



los nodos y y n indican que la máquina termina en un estado de aceptación o rechazo, respectivamente.

Ejemplo 3.9 *Todos los lenguajes del Ejemplo 3.7 son decidibles. En el caso de los programas Java, el formalismo que decide la pertenencia de un input al lenguaje (es decir, la corrección sintáctica de un programa) es el analizador sintáctico que es parte del compilador. Más tarde nos convenceremos de que este analizador se puede implementar con una MT, por lo que el lenguaje de los programas Java también es decidible.*

El ejercicio 3.8 presenta una MT que decide otro lenguaje muy importante en la teoría de los lenguajes formales: su importancia reside en que ningún *autómata finito* es capaz de decidirlo.

La característica fundamental de una MT M que decide un lenguaje recursivo L es que, dado un input w, M es capaz de decir *siempre* cuando $w \in L$ y cuando $w \notin L$. En ningún caso el número de pasos que M tiene que dar para contestar a la pregunta es infinito.

Hecho 3.2 *Una MT que decide $L \subseteq \Sigma^*$ termina para todo input $w \in \Sigma^*$.*

3.3.2 Calcular una función

Como hemos visto en los ejemplos, una Máquina de Turing no se limita a contestar "sí" o "no" a las preguntas: la posibilidad de escribir en una cinta permite calcular funciones. Cuando se dice que un formalismo de computación calcula una función f, el significado es el que nos da la intuición: a partir de un input w representado en cierto lenguaje, la computación produce un output $f(w)$ que corresponde al resultado de aplicar la función a w.

Para comparar diversos formalismos A y B sobre la misma f, lo que hay que hacer es simplemente definir el formato del input y el output para poder comparar el resultado de A y B sobre los mismos input.

Una MT M *calcula* una función $f : \Sigma_0^* \mapsto \Sigma_0^*$ si, para todo $w \in \Sigma_0^*$, el resultado de ejecutar M sobre el input w (que normalmente se representa como $M(w)$) es $f(w)$. Más formalmente, $(s, \triangleright \underline{\sqcup} w) \vdash_M^* (h, \triangleright \underline{\sqcup} f(w))$ donde h es

un estado final. Una función se dice *recursiva*[9] o *calculable* o *efectiva* si existe una Máquina de Turing que la calcula.

Ejemplo 3.10 *La suma y el producto entre números naturales son funciones recursivas porque existen Máquinas de Turing (M_+ y M_*) que las calculan.*

3.3.3 Semidecidir un lenguaje

Una MT M *semidecide* un lenguaje $L \subseteq \Sigma_0^*$ si para todo input $w \in \Sigma_0^*$:

- $w \in L$ si y sólo si M termina en el input w.

Enseguida vemos que esta definición es muy parecida a una parte de la definición de decidibilidad; de hecho, veremos que la decidibilidad de un lenguaje implica su semidecidibilidad.

Ejemplo 3.11 *La MT del Ejemplo 3.3 semidecide el lenguaje $L = \{a^n \mid \exists k.n = 2k{+}1\}$ de las representaciones unarias de los números pares, porque termina si y sólo si el número es par.*

Un lenguaje se dice *semidecidible* o *recursivamente enumerable* si existe una MT que lo semidecide.

Ejemplo 3.12 *El lenguaje $L = \{a^n \mid \exists k.n = 2k{+}1\}$ del Ejemplo 3.11 es recursivamente enumerable porque la máquina del Ejemplo 3.3 lo semidecide. Pero este lenguaje es también recursivo: es fácil diseñar una MT parecida que termina siempre y decide si un número es par o impar.*

A primera vista, podríamos pensar que decidir y semidecidir, ser decidible y ser semidecidible son la misma cosa: al fin y al cabo, parecería que los input no aceptados tengan que ser rechazados. Sin embargo, no es así. Si M decide L, la máquina dará una respuesta para todo input; pero si M semidecide L (sin decidirlo), la no pertenencia de un input al lenguaje no acaba en un rechazo porque M *no terminará* para ese input. Esta diferencia parece poco significativa, pero justamente en estos detalles se juega el núcleo fundamental de la Teoría de la Computabilidad. Por lo que acabamos de decir, el siguiente teorema no vale en la otra dirección.

Teorema 3.1 *Un lenguaje recursivo (decidible) es recursivamente enumerable (semidecidible).*

[9] Que se use el mismo término para lenguajes y para funciones no es casual, ya que se trata de dos conceptos muy relacionados entre ellos.

Demostración. Si L es recursivo, entonces existe M que lo decide, es decir, $M(w)$ termina en y si $w \in L$ y termina en n si $w \notin L$. Se puede construir una MT M' que semidecide L: es igual que M, pero reemplaza el estado n con un nuevo estado q_n, y modifica la δ de M de manera que toda computación que pase por q_n no termina. Por ejemplo, la nueva δ' sería tal que $\delta'(q_n, \sigma) = (\rightarrow, q_n)$ para todo símbolo σ. Es fácil ver que M' semidecide L, por lo que el lenguaje es recursivamente enumerable. □

Teorema 3.2 *El complemento de un lenguaje recursivo es también recursivo.*

Demostración. Sea L un lenguaje recursivo decidido por M, y $\bar{L}$ su complemento. La MT $\bar{M}$ que decide $\bar{L}$ es igual que L pero intercambia los estados finales y y n, así que un input aceptado por M es rechazado por $\bar{M}$, y un input aceptado por $\bar{M}$ es rechazado por M. Por lo tanto, la recursividad de L implica la de $\bar{L}$. □

3.4 Extensiones de las Máquinas de Turing

Hemos visto como componer Máquinas de Turing permite implementar algoritmos más complejos. ¿Hasta dónde podemos llegar con estas herramientas? ¿Existe la posibilidad de ampliar las potencialidades de una MT añadiendo "más hardware" o mejorando el "software"? Nos limitamos a plantear algunas *extensiones* de la definición original de MT que parecen aumentar la clase de funciones que calculan o problemas que (semi)deciden:

- que el alfabeto Σ sea mucho más extenso, incluyendo las cifras de 0 a 9 para poder representar los números en notación decimal, y muchos más símbolos (pero siempre en número finito) si hace falta;
- que la cinta sea infinita en los dos lados, no sólo hacia la derecha;
- que haya más de una cinta (MT con k cintas) y una cabeza en cada cinta;
- que la cinta sea bidimensional (es decir, una superficie plana en lugar de una cinta lineal);
- que cada casilla sea accesible en un paso (Random Access Machine);
- que se incluya el no determinismo.

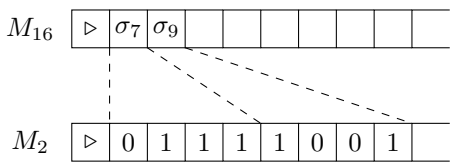
Algunas de estas extensiones no parecen añadir mucha potencia a las Máquinas de Turing; otras, en cambio, se presentan a primera vista como mejoras substanciales. Se puede demostrar que *ninguna* de estas extensiones añade un solo nuevo lenguaje a la clase de los lenguajes decidibles, ni uno solo a la clase de los lenguajes semidecidibles, ni una sola función a la clase de las funciones calculables. En esta sección sólo consideramos algunas extensiones, y únicamente con respecto a la decisión o semidecisión de lenguajes.

3.4.1 Alfabeto con más símbolos

En muchos ejemplos hemos usado máquinas cuyo alfabeto era $\{\triangleright, \llcorner, a\}$; considerando $\triangleright$ como símbolo especial, podemos decir que estas máquinas sólo trabajan con dos símbolos, y podemos cambiar la representación de $\llcorner$ y a por 0 y 1 sin cambiar las características de la máquina de forma significativa.

Ahora, supongamos que exista una máquina M_{16} cuyo alfabeto consiste de 1+16 símbolos: $\triangleright$ más otros 16, llamados $\sigma_0..\sigma_{15}$. Esta máquina decide cierto lenguaje L. Vamos a demostrar que podemos construir una máquina M_2 con sólo 1+2 símbolos $\triangleright$, 0 y 1 que *simula* perfectamente M_{16}, es decir, decide el mismo lenguaje L. Esta demostración se puede extender fácilmente a cualquier número de símbolos, incluso si no es potencia de 2.

La idea es representar cada símbolo de M_{16} con 4 símbolos de M_2, ya que para representar 16 símbolos en binario se necesitan 4 bits: por ejemplo, σ_7 aparecerá como 0111, como indica la figura:



Una configuración en la cinta de M_{16} se podrá simular en M_2 ocupando cuatro veces más espacio; esto puede parecer un problema, pero no lo es al ser la cinta infinita. ¿Cómo trabaja M_2? Intuitivamente, tendrá que simular cualquier paso de M_{16}; tardará en general mucho más, pero esto tampoco es un problema si lo único que nos importa es saber si podrá o no llevar a cabo la misma tarea.

Como siempre un paso de M_{16} supone: (1) leer un símbolo σ_i en la cinta; (2) aplicar δ al símbolo leído y el estado interno q_j, obteniendo (q_k, x); (3) si $x \in \{\sigma_0..\sigma_{15}\}$, escribir el símbolo en la cinta; (4) si $x \in \{\leftarrow, \rightarrow\}$, desplazar la cabeza; (5) cambiar el estado interno a q_k.

El *paso simulado* de M_2 correspondiente al paso de M_{16} comienza cuando la cabeza se encuentra en la casilla donde empieza la representación del símbolo σ_i (la segunda casilla, la sexta, la décima, etc., como se puede ver en la figura). Para todo estado q_j de M_{16}, M_2 necesitará un estado correspondiente s_j donde se encontrará al principio del paso simulado, y 30 estados s_j^b, donde b es una secuencia de bits con longitud de 1 a 4 (es fácil ver que hay 30 secuencias de este tipo) que tienen esta función: cada vez que M_2 se encuentra en un estado s_j^b, sabe que está simulando un paso de M_{16} en q_j y que ha leído hasta el momento la secuencia de bits b. Lo que nos interesa es que cada secuencia de 4 bits que representa σ_h tiene un estado s_j^b correspondiente para todo j: precisamente el estado tal que b es la representación binaria de h. M_2 tiene que encontrarse en este estado después de escanear la secuencia b. Finalmente, otros estados sirven para ejecutar la acción correspondiente a (q_j, σ_h).

La función de transición δ' de M_2 se define en función de δ. Si $\delta(q_j, \sigma_h) = (\leftarrow, q_k)$ entonces tiene que haber una entrada en la tabla de δ' tal que $\delta'(q_j^b, \_)$, donde b es la representación binaria de h, empieza una secuencia de desplazamientos hacia la izquierda y la nueva posición de la cabeza es cuatro casillas más a la izquierda que al principio del paso simulado. Esto se obtiene definiendo otros estados internos en M_2; se trata de algo aburrido pero conceptualmente sencillo. Notamos que el símbolo leído en este caso ya no importa porque esta información está "empotrada" en el estado (es el índice b). El nuevo estado de M_2 será s_k e indica que el nuevo estado de M_{16} es q_k y hay que empezar a simular otro paso. Se han omitido muchos detalles pero el procedimiento debería resultar bastante claro.

Ejemplo 3.13 *Una posible línea de la tabla de* δ

estado	*símbolo*	*nuevo estado*	*acción*
q_3	σ_{15}	$\leftarrow$	q_4

Se simula incluyendo en la tabla de δ' *lo siguiente:*

estado	*símbolo*	*nuevo estado*	*acción*	
s_3	0	s_3^0	$\rightarrow$	
s_3	1	s_3^1	$\rightarrow$	
s_3^0	0	s_3^{00}	$\rightarrow$	
s_3^0	1	s_3^{01}	$\rightarrow$	
s_3^1	0	s_3^{10}	$\rightarrow$	
s_3^1	1	s_3^{11}	$\rightarrow$	
s_3^{00}	0	s_3^{000}	$\rightarrow$	
...				
s_3^{111}	0	s_3^{1110}	0	*ha leído* s_{14}
s_3^{111}	1	s_3^{1111}	1	*ha leído* s_{15}*; debe desplazarse 7 casillas a la izq.*
s_3^{1111}	_	l_4^6	$\leftarrow$	*quedan 6*
l_4^6	_	l_4^5	$\leftarrow$	*quedan 5*
l_4^5	_	l_4^5	$\leftarrow$	*quedan 4*
l_4^4	_	l_4^5	$\leftarrow$	*quedan 3*
l_4^3	_	l_4^5	$\leftarrow$	*quedan 2*
l_4^2	_	l_4^5	$\leftarrow$	*queda 1*
l_4^1	_	s_4	$\leftarrow$	*siguiente paso*

La primera parte de la tabla almacena el símbolo leído: por ejemplo, el estado s_3^{111} *significa que* M_2 *está simulando un paso de* M_{16} *en el estado* q_3 *y hasta el momento ha leído la secuencia* 111 *en su cinta. Cuando* M_2 *se encuentra en* q_3^{1111}*, sabe que la acción correspondiente de* M_{16} *es ir hacia la izquierda, así que tiene que desplazarse 7 casillas hacia la izquierda (3 porque se acaba de desplazar 3 casillas a la derecha y 4 porque cada casilla de* M_{16} *corresponde a 4 de* M_2*), hasta cambiar el estado a* s_4 *y empezar a simular el siguiente paso. Los estado* l_k^n *se acuerdan de que el siguiente estado de* M_{16} *es* q_4*, y que* M_2 *tiene que moverse* n *casillas a la izquierda (existen también estados* r_j^n *que gestionan los desplazamientos a la derecha, etc.).*

M_2 resulta tener muchísimos más estados internos que M_{16}, y es mucho más lenta. Sin embargo, el razonamiento que acabamos de presentar debería hacernos entender que decide exactamente el mismo lenguaje que M_{16}. Como este procedimiento se puede realizar con cualquier MT cuyo alfabeto contiene $16 + 1$ símbolos, podemos concluir que las Máquinas de Turing con $2 + 1$ símbolos son iguales de potentes que las que tienen $16+1$ símbolos y esto vale, en general, para máquinas con alfabetos de cualquier tamaño.

3.4.2 Máquinas de Turing no deterministas

Una MT no determinista se define como una MT normal, pero δ no es una función, sino una *relación*:

$$\delta \subseteq ((K \setminus \{h, e\}) \times \Sigma) \times ((\Sigma \cup \{\leftarrow, \rightarrow\}) \times K)$$

Es decir, a partir de una configuración se puede dar *más de un paso al mismo tiempo.* A primera vista, el no determinismo añade muchísima potencia a una Máquina de Turing porque permite explorar un número ilimitado de computaciones sin necesitar un hardware ilimitado [10]. Para poder comparar estas nuevas máquinas con la definición normal es necesario redefinir los conceptos de aceptación y decisión.

- Una MT no determinista *acepta* un input w si existe al menos un camino desde la configuración inicial hasta una con el estado interno y (no importa si también los hay que terminan en n).
- Una MT no determinista M *decide* un lenguaje L si y sólo si para todo input w
  - existe un número natural n (que depende de M y w) tal que no hay ninguna configuración C alcanzable desde $(s, \triangleright \underline{\sqcup} w)$ en n pasos (es decir, todas las computaciones terminan); y
  - $w \in L$ si y sólo si a partir de $(s, \triangleright \underline{\sqcup} w)$ se llega a $(y, \triangleright u \underline{a} v)$ para ciertos u, v, a (es decir, se acepta w).

Ejemplo 3.14 (El poder del no determinismo) *Consideremos la MT que dice si un número n es compuesto (es decir, no es primo y no es tampoco 0 ni 1). En lugar de intentar secuencialmente con todos los números de* 2 *a* $\sqrt{n}$*, que sería lo que probablemente haría una MT determinista, se puede definir una máquina no determinista que* decide *el lenguaje de los números compuestos (escritos esta vez en notación* binaria*)* intentándolo *al mismo tiempo con todos los factores posibles. Los pasos que daría esta máquina son:*

- *elegir no determinísticamente dos números cuya representación no tenga más bits que la de n, y escribirlos en notación binaria al lado del input;*
- *multiplicar los dos números;*

[10] El no determinismo no se puede implementar en la práctica; aún así, esta sección demuestra que ni siquiera si esto fuera posible las MTs ganarían en potencia.

- *ver si el input y el producto son el mismo número; si lo son, terminar en el estado y; si no lo son, terminar en el estado n.*

Es evidente que esta máquina es "más rápida" que su versión determinista porque se le permite intentar a la vez varias alternativas. Decide el lenguaje porque si el número es compuesto habrá una computación que termina en y entre las muchas que terminan en n, pero es lo que pide la definición de input aceptado.

Después de ver este ejemplo nos preguntamos si añadir el no determinismo a la definición original de las Máquinas de Turing aumenta su capacidad de decidir lenguajes. La respuesta es *no*: las MTs deterministas pueden calcular o decidir *lo mismo* que las no deterministas.

Equivalencia entre determinismo y no determinismo. En este apartado nos hemos centrado en la semidecisión de lenguajes, pero los otros casos (decidir un lenguaje o calcular una función) son muy parecidos. Si una MT no determinista M semidecide un lenguaje, entonces existe una MT determinista M' que hace lo mismo. Esta demostración informal supone que ya sabemos que una MT (determinista) con 2 o más cintas es equivalente en términos de lenguajes semidecididos a una máquina estándar (determinista y con una cinta); no lo hemos demostrado, pero la demostración se encuentra en la literatura [12].

Dado un input w, M' intenta ejecutar sistemáticamente todas las computaciones de M a través de un proceso de *dovetailing*. El punto es que, dada una configuración C, el número de configuraciones posibles en el paso siguiente es limitado por una cantidad fija $r = |K| * (|\Sigma| + 2)$.

La máquina M_d tiene dos cintas: la primera es igual que de M y en la segunda hay n números entre 1 y r: $i_1 i_2..i_n$, que también son parte del input de esta M_d (abajo se explica de dónde vienen). Simulando el j-ésimo paso de M, M_d lee el j-ésimo número i_j de la segunda cinta y ejecuta la alternativa número i_j de M (estará escrita en su δ). Luego va al número siguiente y continúa con el proceso. Al terminar de leer la segunda cinta, termina su ejecución.

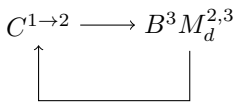
M' usa M_d como componente, y tiene tres cintas:

- la primera nunca cambia y contiene el input;
- la segunda y la tercera se usan para simular M_d para toda secuencia $i_1 i_2..i_n$ de $\{1..r\}^*$ (es decir, todas las computaciones posibles):

  - el input es copiado de la primera cinta a la segunda antes de que M' empiece a simular cada computación;
  - inicialmente la tercera cinta está vacía (primer elemento de $\{1..r\}^*$);

- entre dos simulaciones de M_d, una MT auxiliar genera las secuencias de $\{1..r\}^*$ en orden léxicografico; esto significa que M_d simulará fragmentos cada vez más largos de computaciones de M.

El diagrama global de M' es el siguiente:



- La MT $C^{1\to 2}$ es la que copia el input de la primera a la segunda cinta;
- B^3 genera los elementos de $\{1..r\}^*$ en la tercera cinta;
- $M_d^{2,3}$ es M_d que trabaja en la segunda y en la tercera cinta;

Se puede demostrar que M' para en el input w si y sólo si alguna computación de M lo hace:

- el único modo para que M' termine es que $M_d^{2,3}$ NO termine por haber encontrado un $\llcorner$ en la tercera cinta (es decir, la computación simulada por M_d es finita, y por lo tanto más corta que la secuencia generada en algún momento por B), si no, M continuaría con $C^{1\to 2}$; esto significa que hay una computación de M que termina;
- la otra dirección también vale: si una computación de M termina, entonces tendrá como mucho n pasos, y M_d terminará (no encontrando un $\llcorner$) con un elemento de $\{1..r\}^n$ en la tercera cinta; en este caso, M' también terminará.

Hecho 3.3 *Añadiendo más potencia a una MT (más símbolos, cintas multiples, no-determinismo, más símbolos, etc.) no se aumenta las funciones que se pueden calcular ni los lenguajes que se pueden (semi)decidir, aunque se puede mejorar (y mucho) la eficiencia.*

3.5 Resumen

Resumen 3.1 *Las MTs pueden calcular funciones complejas, aunque de manera muy a menudo ineficiente.*

Resumen 3.2 *Hay Máquinas de Turing cuyas computaciones no terminan (algunas veces, o nunca).*

Resumen 3.3 *Añadiendo más potencia a una MT (añadiendo más cintas, usando una cinta bidimensional, usando más símbolos, incluso incluyendo el no-determinismo) no se aumenta la clase de las funciones que se pueden calcular, aunque se puede mejorar (y mucho, en algunos casos) la eficiencia.*

3.6 Ejercicios

Ejercicio 3.1 *Escribir en forma de tabla la función de transición de la MT que calcula la resta entre dos números $(m - n)$, con la hipótesis que el primer número no sea más pequeño que el segundo.*

Ejercicio 3.2 *Escribir en forma de tabla la función de transición de las MTs M_1 y M_2 que calculan el doble de un número $(2m)$ usando notación unaria (en el caso de M_1) y notación binaria (en el caso de M_2).*

Ejercicio 3.3 *Definir precisa y completamente las siguientes máquinas básicas: R, L, R_σ, L_σ, $R_{\bar{\sigma}}$, $L_{\bar{\sigma}}$, M_σ. Escribir en forma de tabla su función de transición.*

Ejercicio 3.4 *Representar la MT del Ejemplo 3.1 con un diagrama.*

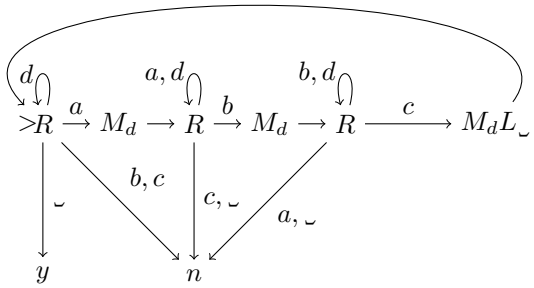
Ejercicio 3.5 *Representar las máquinas del Ejemplo 3.2 con diagramas.*

Ejercicio 3.6 *Representar con diagramas las máquinas M_{decr}, M_c, M_{del} y M_{sh} usadas para implementar M_*.*

Ejercicio 3.7 *Decir lo que hace la siguiente MT:*

$$> R \xrightarrow{a \neq \text{␣}} R \xrightarrow{b \neq \text{␣}} R_{\text{␣}} a R_{\text{␣}} b$$

Ejercicio 3.8 *Determinar el lenguaje decidido por la siguiente MT:*



Ejercicio 3.9 *Dar una MT que decida el lenguaje a^*ba^*b.*

Ejercicio 3.10 *Dar una MT que decida el lenguaje $\{a^*ba^*b\} \cup \{a^*b^*\}$ (la unión de estos dos lenguajes).*

Ejercicio 3.11 *Dar una MT que decida el lenguaje de cadenas de símbolos a y b tales que el número de ocurrencias de a no sea menor que el número de ocurrencias de b. No necesariamente las a's ocurren antes que las b's.*

Ejercicio 3.12 *Dar una MT que semidecida el lenguaje $\{a^nb^n\}$.*

Ejercicio 3.13 *Dar una MT que semidecida el lenguaje $\{a^ncb^n\}$.*

Ejercicio 3.14 *Definir una MT M_4 que decida el lenguaje $\{a^nbc^n \mid n \geq 0\}$ y cuyo alfabeto tenga 4 símbolos $\{a, b, c, \text{␣}\}$ más $\triangleright$. Escribir su función de transición en forma de tabla o diagrama. Definir una MT M_2 con 2 símbolos más $\triangleright$ que decide el mismo lenguaje (cuidado: la representación de una palabra de input en la cinta tendrá que ser distinta, algo parecido a la demostración de la Sección 3.4.1).*

Capítulo 4

Funciones μ-recursivas

Este capítulo describe un nuevo formalismo de computación: unas funciones definidas *por recurrencia*. Primero vamos a introducir una versión "débil" y ver sus limitaciones a la hora de expresar funciones matemáticas o decidir lenguajes. Luego introduciremos el operador μ, que ensancha la clase de funciones que se pueden expresar. Más adelante analizaremos su relación con las Máquinas de Turing.

4.1 Funciones definidas por inducción

Una definición por *recurrencia* describe una *secuencia recursiva*[1]: cada término de la secuencia se define como una función de términos anteriores.

Ejemplo 4.1 (Factorial) *El* factorial *de un número se puede definir por recurrencia con dos ecuaciones: la primera describe el* caso base *(el primer término de la secuencia) mientras que la segunda describe el* caso recursivo *(cómo cada término se define a partir de los anteriores). Sea "!" el nombre de la función: entonces*

$$\begin{cases} !(0) & = & 1 \\ !(n+1) & = & (n+1)\times !(n) \end{cases}$$

[1] Hay una posible confusión: aquí "recursivo" significa "definido por recurrencia", mientras que un lenguaje "recursivo" o "decidible" es uno para el que existe una MT que lo decide (Sec. 3.3.1).

que significa que el primer término de la secuencia, correspondiente al input 0*, es el número* 1*, y el* $(n+1)$*-ésimo término es el anterior multiplicado por* $n+1$*.*

Ejemplo 4.2 (Fibonacci) *Esta función tiene dos casos base que definen los primeros dos términos de la secuencia, y cada término siguiente se define a partir de los dos términos que lo preceden en la secuencia.*

$$\begin{cases} \mathit{fib}\,(0) & = & 1 \\ \mathit{fib}\,(1) & = & 1 \\ \mathit{fib}\,(n+2) & = & \mathit{fib}\,(n+1) + \mathit{fib}\,(n) \end{cases}$$

Para ambas funciones existe una Máquina de Turing que las calcula, luego son funciones *calculables* (o *recursivas*) en este sentido.

4.2 Funciones Primitivas Recursivas

Las funciones que nos interesan son funciones de k números naturales a un número natural ($f : \mathbb{N}^k \mapsto \mathbb{N}$), con $k \geq 0$. Los ejemplos anteriores nos proponen una forma de definir funciones que vamos a profundizar en esta sección.

Aunque no es del todo evidente, la definición de "!" y *fib* se basa en otras funciones más sencillas y en mecanismos para componer funciones. En este caso, las *funciones básicas* que hemos usado son la función constante 1, la suma y el producto. Además, los mecanismos usados para *componer funciones* son

- la *recurrencia*, que permite referirse al resultado de una función en pasos anteriores de la secuencia (como $\mathit{fib}\,(n+1)$ y $\mathit{fib}\,(n)$, que se usan para calcular $\mathit{fib}\,(n+2)$)); y

- la *composición*: en "*fib*", una vez que se tengan los valores $\mathit{fib}\,(n+1)$ y $\mathit{fib}\,(n)$, se componen a través de la función que calcula la suma; es decir, se aplica una función al resultado de otras funciones.

Estas observaciones nos llevan a definir una clase de funciones definidas por recurrencia: las *Funciones Primitivas Recursivas* (FPR). Primero, hay que introducir la notación λ para funciones: $\lambda x, y.z$ es la función que, dados los argumentos x e y, devuelve z. Por ejemplo, la suma se puede escribir como $\lambda n_1, n_2.n_1 + n_2$, mientras que $\lambda x_1, x_2, x_3.x_2$ representa la función que tiene tres argumentos y siempre devuelve el segundo.

Definición 4.1 (Funciones Primitivas Recursivas) *Se define la clase de las Funciones Primitivas Recursivas como la mínima clase $\mathcal{P}$ de funciones de $\mathbb{N}^k$ a $\mathbb{N}$ que incluye:*

- *las* funciones constantes *con cualquier número de argumentos: $\lambda n_1, .., n_k.m$ es la función constante m con aridad k; así tenemos $\lambda n_1, .., n_3.0$, $\lambda n_1, n_2.1$, $\lambda n_1, .., n_4.2$, etc., es decir, una función $const_{n,k}$ para cada constante n y cada valor de aridad k;*
- *el* sucesor *$suc = \lambda x.x + 1$ que devuelve como output el sucesor de su input;*
- *la* identidad *o* proyección*, que selecciona un argumento entre k: $id_{i,k} = \lambda n_1, .., n_k.n_i$ para todo k e i;*

y que es cerrada bajo:

- *la* composición*: si $g_1, .., g_k \in \mathcal{P}$ son funciones de m argumentos, y $h \in \mathcal{P}$ es una función de k argumentos, entonces su composición pertenece a $\mathcal{P}$:*

$$\lambda x_1, .., x_m.h(g_1(x_1, .., x_m), .., g_k(x_1, .., x_m)) \in \mathcal{P}$$

- *la* recursión primitiva*: si $h \in \mathcal{P}$ es una función de $k+1$ argumentos, y $g \in \mathcal{P}$ es una función de $k-1$ argumentos, entonces pertenece a $\mathcal{P}$ la función f de k argumentos definida como:*

$$\begin{cases} f(0, n_2, .., n_k) &= g(n_2, .., n_k) \\ f(n_1 + 1, n_2, .., n_k) &= h(n_1, f(n_1, n_2, .., n_k), n_2, .., n_k) \end{cases}$$

Una clase C de objetos matemáticos es cerrada *bajo una operación cuando aplicar la operación a objetos de C siempre devuleve un objeto de C.*

Una cosa importante es que esta forma de recursión corresponde al bucle **for** de algunos lenguajes como Pascal (importante: no el **for** de C o Java, que es más expresivo). Los ejemplos siguientes nos llevan a ver que "!" y *fib* son, entre otras, primitivas recursivas al encajar perfectamente en esta definición.

Ejemplo 4.3 (Suma) *La suma es primitiva recursiva al poder definirse a partir de suc, composición y recursión primitiva:*

$$\begin{cases} sum\,(0, m) &= id_{1,1}(m) = m \\ sum\,(n + 1, m) &= h(n, sum\,(n, m), m) \end{cases}$$

donde h es $\lambda x, y, z.suc\,(id_{2,3}(x, y, z))$. Las dos líneas de la definición encajan perfectamente con la definicíon de FPR, aunque se ha necesitado una formulación un tanto enrevesada al tener que ignorar dos de los argumentos de h. En los siguientes ejemplos no seremos tan estrictos con la definición de h, pero ya sabemos que se puede componer funciones cuya aridad "no encaja" con la Def. 4.1 usando proyecciones y otros mecanismos bastante intuitivos.

Ejemplo 4.4 (Producto) *El producto es primitivo recursivo al poder definirse a partir de la suma, que también lo es:*

$$\begin{cases} mul\,(0, m) &= const_{0,1}(0) = 0 \\ mul\,(n+1, m) &= sum\,(m, mul\,(n, m)) \end{cases}$$

Esta definición es un ejemplo de lo comentado en el Ejemplo 4.3: sum es una función de dos argumentos mientras que la defínción de recursión primitiva pide una de tres, pero esto se puede arreglar fácilmente definiendo una h de tres argumentos que ignora el tercero aplicando proyecciones.

Ejemplo 4.5 (Inversión 0/1) *Esta función convierte 0 en 1 y todos los demás números en 0.*

$$\begin{cases} inv\,(0) &= const_{1,0} = 1 \\ inv\,(n+1) &= const_{0,2}(n, inv\,(n)) = 0 \end{cases}$$

Ejemplo 4.6 (Diferencia no-negativa)

$$\begin{cases} sub\,(0, m) &= const_{0,1}(0) = 0 \\ sub\,(n+1, m) &= sub\,(n, pred\,(m)) \end{cases}$$

donde pred es el predecesor (Ejercicio 4.3).

Ejemplo 4.7 (Predicado de igualdad)

$$eq\,(m, n) = isZero\,(sum\,(sub\,(m, n)\,, sub\,(n, m)))$$

El último ejemplo nos dice cómo definir *predicados*, que son funciones de $\mathbb{N}^k$ a $\{0, 1\}$. Esto se usa para decidir lenguajes: un predicado primitivo recursivo p decide un lenguaje L si $p(x) = 1$ cuando $x \in L$, y $p(x) = 0$ cuando $x \notin L$.

Una función definida *por casos* a partir de funciones y predicados primitivos recursivos es también primitiva recursiva:

$$f(n_1, .., n_k) = \begin{cases} g(n_1, .., n_k) & \text{si } p(n_1, .., n_k) = 1 \\ h(n_1, .., n_k) & \text{si } p(n_1, .., n_k) = 0 \end{cases}$$

se puede reescribir como $f(n_1,..,n_k) =$

$$sum\left(mul\left(p(n_1,..,n_k),g(n_1,..,n_k)\right),mul\left(inv\left(p(n_1,..,n_k)\right),h(n_1,..,n_k)\right)\right)$$

y con esta nueva herramienta se pueden definir de forma más intuitiva ciertas funciones como algunas de las siguientes.

Ejemplo 4.8 (Resto de división)

$$\begin{cases} rem\,(0,m) & = & const_{0,1}(0) & = & 0 \\ rem\,(n+1,m) & = & \begin{cases} 0 & si\ eq\,(rem\,(n,m),pred\,(m)) = 1 \\ suc\,(rem\,(n,m)) & si\ eq\,(rem\,(n,m),pred\,(m)) = 0 \end{cases} \end{cases}$$

Ejemplo 4.9 (Divisor) *El predicado* $div\,(n,m)$*, que devuelve* 1 *si* n *es divisor de* m *y* 0 *si no lo es, es primitivo recursivo:*

$$div\,(n,m) = inv\,(rem\,(m,n))$$

Lema 4.1 (Maximización "bounded" (acotada)) *Sea* $f : \mathbb{N}^{k+1} \mapsto \mathbb{N}$ *una función. Para todo* $y \in \mathbb{N}$*, se define*

$$A_f(y,\bar{x}) = \{z \in \mathbb{N} \mid z \leq y \wedge f(z,\bar{x}) = 0\}$$

También se define

$$f_{bmax}(y,\bar{x}) = \begin{cases} \max A_f(y,\bar{x}) & si\ A_f(y,\bar{x}) \neq \emptyset \\ 0 & si\ A_f(y,\bar{x}) = \emptyset \end{cases}$$

Si f *es una función primitiva recursiva, entonces* f_{bmax} *también lo es.*

Demostración. f_{bmax} se puede escribir como

$$\begin{cases} f_{bmax}(0,\bar{x}) & = & const_{0,k}(\bar{x}) = 0 \\ f_{bmax}(n+1,\bar{x}) & = & \begin{cases} suc\,(n) & \text{si } f(suc\,(n),\bar{x}) = 0 \\ f_{bmax}(n,\bar{x}) & \text{si } f(suc\,(n),\bar{x}) \neq 0 \end{cases} \end{cases}$$

Así que su computación se reduce a calcular f como mucho y veces hasta que el resultado sea 0. Si esto nunca sucede, el resultado final es 0. Notamos que esta definición de f_{bmax} entra dentro del esquema general de recursión primitiva. □

Concretamente, $f_{bmax}(y,\bar{x})$ viene a ser la maximización "bounded" de f, es decir, el máximo z entre 0 e y tal que $f(z,\bar{x})$ es igual a 0. Es importante notar que la limitación "entre 0 e y" es fundamental para poder decir que f_{bmax} es primitiva recursiva.

Ejemplo 4.10 (Logaritmo entero) *También es primitiva recursiva la función* $lo\,(n,m)$ *que devuelve el máximo numero* k *tal que* $div\left(n^k,m\right)$*, es decir, el máximo exponente que aplicado a* n *da un número divisor de* m*. Es decir,* $lo\,(n,m)$ *es* $f_{bmax}(m,n,m)$ *donde* $f(k,n,m)$ *es* $inv\left(div\left(n^k,m\right)\right)$*. Notamos que el primer argumento de* f_{bmax} *es* m*, es decir, el máximo exponente* k *lo buscamos en* $\{1,..,m\}$ *(por eso se usa la maximización "bounded"). Por ejemplo,* $lo\,(2,18)=1$ *porque ni* 2^2 *ni* 2^3 *ni* 2^4 *son divisores de 18.*

Ejemplo 4.11 (Factorial) *La definición de "!" usa la constante 1, el producto, la composición y la recursión primitiva* [2]*. El esquema de recursión utilizado por "!" se reconduce a la recursión primitiva donde*

- f *es la misma "!", con aridad* $k=1$*;*
- *el caso base* g *es la función constante* 1 *con aridad* 0*;*
- h *es la función binaria* $\lambda n_1,n_2.mul\,(suc\,(n_1),n_2)$*, es decir, la función que suma* 1 *al primer argumento y lo multiplica por el segundo.*

Por lo tanto, "!" es una Función Primitiva Recursiva.

Ejemplo 4.12 (Fibonacci) *Con estos resultados podemos ver que fib también es primitiva recursiva. Ya teníamos una definición por recurrencia de esta función pero no encajaba en el esquema de recursión primitiva porque había* dos *casos base, así que necesitamos otra definición equivalente que sí encaje en la definición. Sea* fib_0 *la función definida como*

$$\begin{cases} fib_0(0) &= 6 \\ fib_0(n+1) &= mul(exp\,(2,lo\,(3,fib_0(n)))\,, \\ & \qquad exp\,(3,sum\,(lo\,(3,fib_0(n))\,,lo\,(2,fib_0(n))))) \\ & = 2^{lo(3,fib_0(n))}\times 3^{lo(3,fib_0(n))+lo(2,fib_0(n))} \end{cases}$$

donde la última línea es una forma más comprensible de escribir el caso $n+1$*. Esta función es primitiva recursiva porque sus componentes fundamentales lo son. Entonces* $fib\,(n)$ *se define como* $lo\,(2,fib_0(n))$ *y resulta ser primitiva recursiva. Este ejemplo nos sirve para darnos cuenta de que con composición y recursión primitiva se puede definir funciones no triviales, incluso una como fib que en su definición más intuitiva tiene dos casos base. Claro está, hace falta tiempo y unas cuantas pruebas para convencernos de que* $lo\,(2,fib_0(n))$ *es realmente un término de la secuencia de Fibonacci.*

[2] No sería correcto afirmar que esta definición usa la suma, aunque el símbolo "+" sí aparece en ella; más bien este símbolo es una componente del esquema de recursión primitiva.

4.3 Límites de las FPRs

Como acabamos de ver, usando los ingredientes de las Funciones Primitivas Recursivas se pueden definir muchas funciones interesantes y no triviales. Además, un análisis detallado de su definición nos indica que sólo definen funciones *totales*, es decir, que devuelven un resultado para todo input, y lo devuelven *en un número finito de pasos de computación*, donde un paso de computación se puede definir informalmente como una "llamada a función"[3].

Ejemplo 4.13 *El resultado de* $sum\,(3,2)$ *se calcula en 11 pasos:*

$$
\begin{aligned}
sum\,(3,2) &=^1 suc\,(id_{2,3}(2, sum\,(2,2)\,,2)) \\
&=^2 suc\,(id_{2,3}(2, suc\,(id_{2,3}(1, sum\,(1,2)\,,2))\,,2)) \\
&=^3 suc\,(id_{2,3}(2, suc\,(id_{2,3}(1, suc\,(id_{2,3}(0, sum\,(0,2)\,,2))\,,2))\,,2))\,,2)) \\
&=^4 suc\,(id_{2,3}(2, suc\,(id_{2,3}(1, suc\,(id_{2,3}(0, id_{1,1}(2),2))\,,2))\,,2))\,,2)) \\
&=^5 suc\,(id_{2,3}(2, suc\,(id_{2,3}(1, suc\,(id_{2,3}(0,2,2))\,,2))\,,2))\,,2)) \\
&=^6 suc\,(id_{2,3}(2, suc\,(id_{2,3}(1, suc\,(2)\,,2))\,,2))\,,2)) \\
&=^7 suc\,(id_{2,3}(2, suc\,(id_{2,3}(1,3,2))\,,2))\,,2)) \\
&=^8 suc\,(id_{2,3}(2, suc\,(3)\,,2))\,,2)) \\
&=^9 suc\,(id_{2,3}(2,4,2))\,,2)) \\
&=^{10} suc\,(4) \\
&=^{11} 5
\end{aligned}
$$

En cada paso se aplica la definición de alguna función a un input concreto.

Esto no sucede con las Máquinas de Turing, porque hay casos en que una MT no termina dado cierto input, es decir, expresa una función *parcial*. En general, no es difícil ver que cualquier FPR puede ser calculada por una MT:

Hecho 4.1 *Todas las Funciones Primitivas Recursivas son calculables.*

Hecho 4.2 *Todas las Funciones Primitivas Recursivas son funciones totales al necesitarse un número finito de pasos de computación para calcularlas.*

Entonces nos preguntamos: ¿hemos encontrado el formalismo de computación perfecto, que puede calcular todo lo *calculable* y encima con la garantía de que va a tardar un tiempo finito (es decir ejecutar un número finito de

[3]Una posible objeción es: ¿qué pasa si definimos una f en función de g y g en función de f, así que se "llaman" recíprocamente? Esta situación se llama *recursión mutua* y no es posible con las FPRs porque al definir f en función de g necesito tener *ya* la definición completa de g, que no puede apoyarse en la de f porque es lo que todavía no está definido.

pasos de computación) en cada computación? Desafortunadamente, no es así: usando el método de *diagonalización*, se demuestra que las Funciones Primitivas Recursivas *no* pueden calcular todas las funciones calculables totales, es decir, hay algunas que, aún siendo funciones calculables totales, se les escapan.

Para demostrar que hay funciones calculables totales que no son FPRs, vamos a limitarnos a funciones con un argumento, de naturales a naturales ($\mathbb{N} \mapsto \mathbb{N}$); el resultado es generalizable a funciones con más argumentos.

Observación 4.1 *Dado un alfabeto Σ, las sequencias finitas de símbolos de Σ se pueden* enumerar *una tras otra sin dejar fuera ninguna:*

- *primero se enumera la única secuencia de 0 símbolos;*
- *luego las $|\Sigma|$ sequencias de 1 símbolo;*
- *luego las $|\Sigma|^2$ sequencias de 2 símbolos;*
- *luego las $|\Sigma|^3$ sequencias de 3 símbolos;*
- *etc.*

Por lo tanto, estas secuencias son enumerables[4] *y a cada una se le puede asignar de forma unívoca y sin repeticiones un número natural.*

Ejemplo 4.14 *Sea $\Sigma = \{a, b, c, d\}$: siguiendo el orden alfabético se obtendrían las secuencias*

$$\begin{array}{l} \varepsilon(\textit{secuencia vacía}), \\ a, b, c, d, \\ aa, ab, ac, ad, ba, bb, bc, bd, ca, cb, cc, cd, da, db, dc, dd, \\ aaa, aab, aac, aad, aba, abb, abc, abd, aca, \textit{etc.} \end{array}$$

Por lo tanto, a la secuencia ε corresponde el 0, a "a" corresponde el 1, a "c" corresponde el 3, a "dc" corresponde el 19, etc.

Observación 4.2 *Las FPRs son* enumerables*, es decir, a cada una se le puede asignar un número natural sin repetir; son enumerables porque*

- *la definición de una función (es decir, su descripción gráfica) no es otra cosa que una secuencia* finita *de símbolos de cierto alfabeto* finito*;*

[4] Su *cardinalidad* viene a ser la misma que la de $\mathbb{N}$. El Capítulo 7 introduce más ideas sobre la cardinalidad de un conjunto.

- *dado un alfabeto Σ, se puede enumerar las secuencias de símbolos de Σ según un orden, como explicado en la Observación 4.1, y quitar (por medio de un fácil control* sintáctico*) las que no describen funciones primitivas recursivas (p.ej., "vebf&39&yvpw%* uibdfp" parece ser una de ellas).*

Entonces se trata de generar la secuencia (en principio infinita) de funciones $f_0, f_1, .., f_n, ..$ que cubren todas las Funciones Primitivas Recursivas.

Podemos representar la n-ésima función f_n como la lista (infinita) de todos los valores que devuelve: $\langle f_n(0), f_n(1), .., f_n(17), ..\rangle$.

Ejemplo 4.15 *Si f_{32} es la función que calcula el cuadrado de un número, entonces su representación sería $\langle 0, 1, 4, 9, 16, 25, 36, 49, ..\rangle$.*

Si representamos de esta forma todas las funciones de la secuencia obtenemos una tabla con un número infinito (enumerable) de filas con un número infinito (enumerable) de elementos:

$$\begin{array}{ccccccc} f_0(0) & f_0(1) & f_0(2) & ... & f_0(7) & ... \\ f_1(0) & f_1(1) & f_1(2) & ... & f_1(7) & ... \\ f_2(0) & f_2(1) & f_2(2) & ... & f_2(7) & ... \\ ... & ... & ... & ... & ... & \\ f_{14}(0) & f_{14}(1) & f_{14}(2) & ... & f_{14}(7) & ... \\ ... & ... & ... & ... & ... & \end{array}$$

donde el elemento que está en la fila n, columna m es el resultado de calcular $f_n(m)$. En la tabla no hay "huecos" porque todas las FPRs son totales (es decir, definidas para todo input).

El siguiente paso es definir una función f (también total) tal que, para todo n, $f(n) = f_n(n)$; es decir, f devuelve los valores de la *diagonal* de la tabla. Nos preguntamos si f es primitiva recursiva, es decir, si los valores en la diagonal son iguales (y aparecen en el mismo orden) que una cierta f_n representada en alguna línea de la tabla.

En general, no tenemos una respuesta: podría ser que la diagonal fuera exactamente igual a alguna de las líneas. Por otro lado, si se define otra función f' tal que para todo n, $f'(n) = f_n(n) + 1$, es decir, cada uno de sus elementos es el sucesor del correspondiente elemento de la diagonal, obtenemos que f' es

- total (porque f y "+1" también lo son);

- calculable, porque para todo input n basta calcular $f_n(n)+1$ y $f_n(n)$ es calculable al ser f_n una función primitiva recursiva, y "+1" es el sucesor, que también es calculable;

- distinta de cualquier f_n; es decir, no hay una línea en la tabla cuyos valores sean todos iguales a los de la diagonal más 1: toda f_n discrepará de f' por lo menos en el "puesto" n, así que las dos funciones no pueden ser iguales.

Ahora, el hecho de que f' sea distinta de cualquier f_n implica que *no* es primitiva recursiva, porque la hipótesis inicial era que las líneas de la tabla las representan todas.

La conclusión es que existen funciones calculables totales de $\mathbb{N}$ a $\mathbb{N}$ que *no* son primitivas recursivas. Por lo tanto, el formalismo de las Funciones Primitivas Recursivas no captura todo lo interesante acerca de la idea de computación. Más en general:

- una función con las características de f' se puede encontrar independientemente del orden de las líneas de la tabla; es decir, cambiando la enumeración de las FPRs se generaría otra función f' que, aunque distinta de la anterior, seguiría cumpliendo con las tres condiciones que acabamos de enunciar;

- este método de diagonalización se puede aplicar a *cualquier* formalismo que sólo define funciones totales.

Hecho 4.3 *No se puede definir un formalismo de computación que incluya sólo funciones totales y que las incluya todas.*

Ejemplo 4.16 *El típico ejemplo de función calculable y total pero no primitiva recursiva es la función de Ackermann: su definición usa una* doble recursión *que no se puede expresar con los mecanismos de la recursión primitiva.*

$$\begin{aligned} A(0,0,y) &= y \\ A(0,x+1,y) &= A(0,x,y)+1 \\ A(1,0,y) &= 0 \\ A(z+2,0,y) &= 1 \\ A(z+1,x+1,y) &= A(z,A(z+1,x,y),y) \end{aligned}$$

En esta formulación, si el primer argumento es 0 entonces A calcula la suma de los otros dos; si es 1 calcula el producto; si es 2 calcula el exponencial; si es más de 2, calcula generalizaciones cada vez más complejas del exponencial.

Hecho 4.4 *Las Funciones Primitivas Recursivas no definen todas las funciones calculables totales.*

La diagonalización es una herramienta matemática para hacer demostraciones que fue usada por el matemático Georg Cantor en 1891 para demostrar que hay conjuntos de números (como los números reales) que no son enumerables. Después de aplicar este método a las FPR, podríamos pensar que se puede aplicar a *cualquier* formalismo de computación, por lo que siempre podemos encontrar funciones calculables que no se pueden calcular con dicho formalismo. Sin embargo no es así: la diagonalización funciona para los formalismos que sólo definen funciones *totales*, es decir, cuyas computaciones terminan siempre.

4.4 Funciones μ-Recursivas

Acabamos de demostrar que las Funciones Primitivas Recursivas sólo calculan funciones calculables[5] totales pero no calculan *todas* las calculables totales: incluso si nos limitamos a las funciones totales, las MTs hacen cosas que estas funciones no pueden hacer.

Hecho 4.5 *Las Máquinas de Turing tampoco calculan todas las funciones totales (sólo las calculables, que son una pequeña parte), pero calculan funciones totales que las Funciones Primitivas Recursivas no pueden calcular.*

Para solucionar este límite de las funciones Primitivas Recursivas introducimos una nueva forma de definir funciones, que se añade a la composición y la recursión primitiva (Def. 4.1): la *minimización*. Dada una función g de $k+1$ argumentos, la función f de k argumentos definida como:

$$f(n_1,..,n_k) = \begin{cases} \text{el mínimo } m \text{ tal que } g(n_1,..,n_k,m) = 1 & \text{si existe} \\ 0 & \text{si no existe} \end{cases}$$

se llama *minimización* de g, se denota normalmente como $\mu m\,[g(n_1,..,n_k,m) = 1]$, y se lee como "el mínimo m tal que $g(n_1,..,n_k,m)$ es igual a 1".

[5]Otra vez, estamos asumiendo haber demostrado que no hay nada que la Funciones Primitivas Recursivas hacen y que las MTs no hacen.

Una cosa importante e intuitiva es que esta última forma de combinar funciones corresponde al bucle **while** (lo veremos más adelante). No existe un método para calcular (siempre en un número finito de pasos) esta minimización, aunque sepamos cómo calcular g para todo input: si m no existe, entonces la única forma de saberlo será intentar con todos los números naturales. Esto corresponde al programa imperativo

```
1 m := 0;
2 while (¬(g(n1,..,nk,m)=1)) do m := m+1;
3 return m;
```

Una función se dice *μ-Recursiva* si se define a partir de las funciones básicas, de la composición, la recursión y la minimización.

Definición 4.2 (Funciones μ-Recursivas) *Se define la clase de las Funciones μ-Recursivas*[6] *como la mínima clase $\mathcal{M}$ de funciones de $\mathbb{N}^k$ a $\mathbb{N}$ que incluye:*

- *las* funciones constantes *$const_{n,k}$ con cualquier número de argumentos;*
- *el* sucesor *$suc = \lambda x.x + 1$;*
- *la* identidad *o* proyección *$id_{i,k}$;*

y que es cerrada bajo:

- *la* composición*;*
- *la* recursión primitiva*;*
- *la* minimización*: si $g \in \mathcal{M}$ es una función de $k+1$ argumentos, pertenece a $\mathcal{M}$ la función f de k argumentos definida como:*

$$f(n_1,..,n_k) = \begin{cases} \textit{el mínimo } m \textit{ tal que } g(n_1,..,n_k,m) = 1 & \textit{si existe} \\ \textit{indefinido} & \textit{si no existe} \end{cases}$$

Ejemplo 4.17 *Usando el mecanismo de la minimización se puede definir el logaritmo* $\log_{m+2}(n+1)$*:*

$$log\,(m,n) = \mu p\ [mayorIgual\,((m+2)^p, n+1)]$$

que viene a ser el mínimo exponente p tal que $(m+2)^p \geq n+1$. Notamos que el logaritmo se puede también calcular con una FPR (es decir, sin usar la minimización).

[6] Otros textos introducen las Funciones μ-Recursivas como funciones totales (porque g siempre vale 1 para algún m) y desarrollan los siguientes argumentos de forma ligeramente distinta. El enfoque dado en este libro nos parece más sencillo e igual de riguroso.

4.4.1 Aplicación de la diagonalización

Nos preguntamos si podemos utilizar el método de la diagonalización para hallar una función calculable que no sea μ-recursiva.

Al fin y al cabo estas funciones siguen siendo definidas por cadenas finitas de símbolos; por lo tanto, enumerando estas funciones tal y como se hizo con las Primitivas Recursivas, se podría aplicar la diagonalización y encontrar una función calculable total que no es μ-recursiva. Pero esto no funciona: enumerar todas las funciones en una tabla no llena todo el espacio porque las funciones μ-recursivas son *parciales*.

Hecho 4.6 *Las funciones μ-recursivas, tal y como se definen en este libro, son funciones* parciales.

¿Por que estas funciones son parciales? Porque su definición incluye la posibilidad de devolver un valor *indefinido* si, para ciertos valores $n_1..n_k$, el valor de $g(n_1, .., n_k, m)$ no es 0 para ningún m, cosa que puede suceder perfectamente. Por lo tanto, en la tabla hay "agujeros", y la función f' construida sobre la diagonal viene a ser una función parcial: por ejemplo

$$f'(n) = \begin{cases} f_n(n) + 1 & \text{si } f_n(n) \text{ está definida} \\ \text{indefinida} & \text{si } f_n(n) \text{ no está definida} \end{cases}$$

Supongamos entonces que f' sea representada por el m-ésimo algoritmo (es decir, que sea calculable): entonces, no se puede concluir el resultado contradictorio $f_m(m) = f(m) = f_m(m) + 1$ porque f_m puede ser parcial.

Hecho 4.7 *La diagonalización no se aplica a las Funciones μ-Recursivas, es decir, no se puede demostrar que existen funciones calculables totales que no son μ-Recursivas.*

Última objeción: y ¿si se extendiesen todas las funciones parciales con ceros para que se conviertan en funciones totales? Desafortunadamente, esto no se puede hacer porque no siempre hay un algoritmo que calcula la función extendida.

4.5 Resumen

Resumen 4.1 *Hay dos significados de la palabra "recursivo": (1) calculable por una MT (se aplica a una función matemática); y (2) definido por recurrencia, como las funciones primitivas recursivas y las μ-recursivas. Más adelante se verá que los dos significados están relacionados.*

Resumen 4.2 *Las Funciones Primitivas Recursivas sólo definen funciones calculables totales, pero no definen todas las calculables totales. Por lo tanto, las MTs hacen cosas que las FPR no pueden hacer.*

Resumen 4.3 *Las Funciones μ-Recursivas no caen bajo los golpes de la diagonalización.*

4.6 Ejercicios

Ejercicio 4.1 *Definir una MT que calcule la función* ! *(factorial) y otra que calcule la función de Fibonacci.*

Ejercicio 4.2 *Modificar la definición del producto (Ejemplo 4.4) para que encaje* perfectamente *en la Def. 4.1 (véase la discusión de los Ejemplos 4.3 y 4.4).*

Ejercicio 4.3 *Demostrar que el predecesor pred (con* $pred\,(0) = 0$*) es primitivo recursivo.*

Ejercicio 4.4 *Demostrar que el exponencial (función* $exp\,(n, m) = n^m$*) es primitivo recursivo partiendo del hecho que suma y producto lo son.*

Ejercicio 4.5 *Demostrar que el predicado "es cero" (devuelve 1 en 0 y 0 en los demás inputs) es primitivo recursivo.*

Ejercicio 4.6 *¿Es primitiva recursiva la función matemática f cuyo resultado se expresa como*

$$\begin{cases} f(0) &= 0 \\ f(n+1) &= g(n) \end{cases} \qquad \begin{cases} g(0) &= 0 \\ g(n+1) &= f(n) \end{cases}$$

Ejercicio 4.7 *¿Es primitiva recursiva la función matemática f cuyo resultado se expresa como*

$$\begin{cases} f(0) &= 0 \\ f(n+1) &= g(n) \end{cases} \qquad \begin{cases} g(0) &= 0 \\ g(n+1) &= f(n+2) \end{cases}$$

Ejercicio 4.8 *Demostrar que la función f del Ejercicio 4.7 es μ-Recursiva.*

Ejercicio 4.9 *(Para valientes) Definir la función de Ackermann como función μ-Recursiva.*

Ejercicio 4.10 *Simular a través de unas MTs el mecanismo de composición de funciones usado por las Funciones μ-Recursivas.*

Ejercicio 4.11 *Simular a través de unas MTs el mecanismo de recursión primitiva usado por las Funciones μ-Recursivas.*

Ejercicio 4.12 *Simular a través de unas MTs el mecanismo de minimización usado por las Funciones μ-Recursivas.*

Capítulo 5

Lenguages FOR y WHILE

5.1 El Lenguaje FOR

En este sección se define el *lenguaje* FOR. Se trata de un lenguaje de programación imperativo muy sencillo que trabaja con números enteros e incluye las operaciones básicas, un bucle **for** y rutinas, pero sin posibilidad de recursión. La *gramática* (el conjunto de reglas para generar programas sintácticamente correctos) de este lenguaje es:

$$
\begin{array}{lcl}
\mathcal{E} & ::= & \mathcal{A} \mid \mathcal{B} \\
\mathcal{A} & ::= & n \mid \mathcal{X} \mid \mathcal{A}+\mathcal{A} \mid \mathcal{A}\times\mathcal{A} \mid \mathcal{A}-\mathcal{A} \mid \mathcal{A}/\mathcal{A} \mid \mathcal{A}\%\mathcal{A} \mid \mathcal{X}(\mathcal{E},..,\mathcal{E}) \\
\mathcal{B} & ::= & \textsf{true} \mid \textsf{false} \mid \mathcal{X} \mid \mathcal{A}=\mathcal{A} \mid \mathcal{A}<\mathcal{A} \mid \neg\mathcal{B} \mid \mathcal{B}\vee\mathcal{B} \mid \mathcal{B}\wedge\mathcal{B} \\
\mathcal{C} & ::= & \textsf{skip} \mid \mathcal{X}:=\mathcal{E} \mid \mathcal{C};\mathcal{C} \mid \{\mathcal{C}\} \mid \textsf{if } \mathcal{B} \textsf{ then } \mathcal{C} \ [\textsf{else } \mathcal{C}] \\
 & \mid & \textsf{for } \mathcal{X}=\mathcal{A} \textsf{ to } \mathcal{A} \textsf{ do } \mathcal{C} \mid \textsf{return } \mathcal{E} \\
\mathcal{F} & ::= & \textsf{proc } \mathcal{X}(\mathcal{X}:\mathcal{T},..,\mathcal{X}:\mathcal{T}) \ [\textsf{returns } \mathcal{T}] \ \{[\mathcal{V}] \ \mathcal{C}\} \\
\mathcal{X} & ::= & x_1 \mid x_2 \mid ... \mid f \mid g \mid ... \\
\mathcal{V} & ::= & \textsf{var } \mathcal{X}:\mathcal{T},..,\mathcal{X}:\mathcal{T}; \\
\mathcal{T} & ::= & \textsf{int} \mid \textsf{bool} \\
\mathcal{P} & ::= & [\mathcal{F}..\mathcal{F}] \ [\mathcal{V}] \ \mathcal{C}
\end{array}
$$

Esta gramática es bastante intuitiva. El símbolo que precede "::=" es el nombre de una *categoría sintáctica*, y lo que sigue es cómo puede ser su estructura sintáctica; el símbolo "|" es una disyuntiva. Por ejemplo, la primera línea define las expresiones $\mathcal{E}$, que pueden ser aritméticas ($\mathcal{A}$) o booleanas ($\mathcal{B}$). Las expresiones aritméticas pueden ser constantes numéricas, variables,

combinaciones de otras expresiones usando operaciones aritméticas o, finalmente, llamadas a rutinas. Las expresiones booleanas son parecidas. La categoría $\mathcal{C}$ describe los comandos o sentencias, que pueden ser: comando vacío (**skip**), asignación de un valor a variable, concatenación de comandos, sentencia condicional, bucle de tipo **for** o devolución de una expresión. $\mathcal{F}$ son las declaraciones de rutinas y empiezan por **proc**, seguido por el nombre de la rutina (o procedimiento), los parámetros formales con sus tipos ($\mathcal{T}$), el tipo de retorno (opcional, porque la rutina podría no devolver nada) y el cuerpo, que es un comando. $\mathcal{X}$ denota los identificadores, tanto de variables como de funciones; $\mathcal{V}$ son las declaraciones de listas de variables, cada una con su tipo. Finalmente, $\mathcal{P}$ es un programa completo, que consiste en unas declaraciones de rutinas, unas declaraciones de variables y una *sentencia principal* por la que empieza la ejecución, y que básicamente es la encargada de proporcionar valores concretos para los parámetros de las rutinas. Los paréntesis "[ ]" indican que el elemento que encierran puede aparecer o no (por ejemplo, un if-then-else no tiene por qué tener la rama "else" si ésta no hace nada). Las rutinas se declaran como funciones y pueden devolver o no un valor entero o booleano con el comando **return**. Se supone que existe un control de tipos que asegura que un programa es bien tipado. Los comentarios de una línea empiezan por //.

Ejemplo 5.1 *El siguiente es un programa correcto en el lenguaje* FOR *que calcula (de forma poco eficiente) la suma de los números primos entre* 5 *y* 10.

```
1  proc sumaPrimos(n1:int,n2:int) returns int {
2    var x:int;
3    var m:int;
4    x := 0;
5    for m=n1 to n2 do {
6      if esPrimo(m) then x := x+m
7    };
8    return x
9  }
10
11 proc esPrimo(m:int) returns bool {
12   var p:int;
13   var f1:int;
14   var f2:int;
15   if (m=0) then return false;
16   if (m=1 ∨ m=2) then return true;
17   p := true;
18   for f1=2 to m do {
```

```
19      for f2=2 to m do {
20        if (f1*f2=m) then p := false
21      }
22    };
23    return p
24 }
25
26 sumaPrimos(5,10); // (comando principal)
```

En lo siguiente, si esto no genera ambigüedad, se utilizaá este lenguaje de forma bastante libre. Por ejemplo, a veces se omitirá la sentencia principal al tener entendido que será simplemente el código que llama una rutina tras asignar valores concretos a sus parámetros (es el caso del ejemplo anterior).

El único bucle que se puede usar es el *for*. Es muy importante entender que este bucle *no* es igual al de C o Java, sino que es igual más bien al *for* de lenguajes como Pascal. La diferencia (muy grande) es que *antes de empezar la primera iteración* se sabe cuántas iteraciones se van a dar. Esto sucede porque las dos expresiones antes y después de `to` se evaluan *antes* de ejecutar el bucle, y definen el número de iteraciones; por ejemplo, **for** `x=1 to 10` **do** C ejecuta el cuerpo C del bucle 10 veces, mientras que **for** `x=1 to 0` **do** C no lo ejecuta nunca. Cada vez que entramos en C, la variable `x` tendrá un valor distinto (de 1 a 10, en el primer bucle). En el bucle **for** `f1=2 to m` **do** C, a tiempo de compilación no se sabe cuántas iteraciones va a haber, pero se sabrá a tiempo de ejecución antes que el bucle empiece a ejecutarse. Es fácil caer en la cuenta de que ni el **for** ni el **while** de C o Java funcionan de esta forma.

Hecho 5.1 *En un programa* FOR*, se sabe cuántas iteraciones va a dar un bucle* **for** *antes de empezar a ejecutarlo (aunque no necesariamente antes de empezar a ejecutar el programa principal).*

Otra característica de este lenguaje es que no admite *recursión*: es decir, una función no puede llamarse a sí misma, y si una función `f` llama a `g`, entonces `g` no puede llamar a `f` ni siqueira indirectamente. A nivel de potencia de cálculo, la recursión no añade nada a un lenguaje imperativo; en general, su interés reside en el hecho de que permite implementar ciertos algoritmos de forma elegante (por ejemplo el factorial o la serie de Fibonacci). Dado un program con recursión existe un programa equivalente sin ella; por lo tanto, no hay que preocuparse si no está permitida.

Ejemplo 5.2 *El siguiente programa, que incluye una llamada recursiva,*

```
proc f(n1:T1,..,nk:Tk) returns T {
  if (n1=0) then return g(n2,..,nk)
  else return h(n1-1,f(n1-1,n2,..,nk),n2,..,nk)
}
```

se transforma de forma sistemática en un programa equivalente sin recursión:

```
proc f(n1:T1,..,nk:Tk) returns T {
  var ret:T;
  var i:int;
  ret := g(n2,..,nk);
  for i=0 to m-1 do ret := h(i,ret,n2,..,nk);
  return ret;
}
```

Es fácil darse cuenta de que este programa implementa un mecanismo bastante familiar (Sección 4.2). Un ejemplo más concreto es el siguiente (es el resto no negativo; la recursión se hace sobre el segundo argumento pero esto no supone ninguna diferencia notable):

```
proc sub(n:int,m:int) returns int {
  if (m=0) then return n
  else return pred(sub(n,m-1))
}
```

se puede transformar en

```
proc sub(n:int,m:int) returns int {
  var ret:int;
  var i:int;
  ret := n;
  for i=0 to m-1 do ret := pred(ret);
  return ret;
}
```

donde pred *implementa la función "predecesor" no negativo (Ejercicio 4.3).*

Hecho 5.2 *Dado un programa imperativo que usa recursión, existe siempre un programa equivalente sin recursión (sólo con iteración).*

También hay otra observación acerca de los programas no recursivos:

Hecho 5.3 *Dado un program imperativo en el que no se usa recursión, se puede reorganizar el código para que todo el programa sea una única rutina (en este caso, una única función).*

Por lo tanto, ni la recursión ni la presencia de rutinas ensancha la clase de funciones o algoritmo que se pueden calcular usando cierto lenguaje.

5.1.1 Lenguaje FOR vs. Funciones Primitivas Recursivas

El siguiente teorema afirma que cualquier cosas que las FPRs pueden calcular, los programas FOR también pueden calcularla.

Teorema 5.1 *Dada una función de k argumentos $f : \mathbb{N}^k \mapsto \mathbb{N}$ que se puede representar como Función Primitiva Recursiva, existe una rutina escrita en el lenguaje* FOR *que recibe k inputs $n_1..n_k$ y devuelve $f(n_1..n_k)$.*

Demostración. Se trata de (1) demostrar que las Funciones Primitivas Recursivas básicas se pueden implementar como rutinas FOR; y (2) que los dos mecanismos de combinar funciones (composición y recursión primitiva) también se representan con programas FOR, es decir, (2a) dadas $g_1, .., g_k$ y h primitivas recursivas, e implementadas por programas FOR funG1 .. funG1 y funH, también existe un programa FOR que implementa la función

$$f = \lambda x_1..x_m.h(g_1(x_1..x_m), .., g_k(x_1..x_m))$$

y (2b) dadas g y h primitivas recursivas, e implementadas por programas FOR funG y funH, también existe un programa FOR que implementa la función f definida como

$$\begin{cases} f(0, n_2, .., n_k) &= g(n_2, .., n_k) \\ f(n_1 + 1, n_2, .., n_k) &= h(n_1, f(n_1, n_2, .., n_k), n_2, .., n_k) \end{cases}$$

Las funciones constantes se implementan fácilmente con rutinas del tipo

```
proc funConst_nk(n1:int,..,nk:int) returns int { return 7;
    }
```

La función sucesor también es muy fácil:

```
proc funSuc(n:int) returns int { return n+1; }
```

Y también las identidades o proyecciones:

```
proc funId_ik(n1:int,..,nk:int) returns int { return ni; }
```

La composición se obtiene fácilmente en el lenguaje FOR: dadas unas rutinas funG1..funGk y funH, la rutina funF buscada se obtiene simplemente llamando a

las demás en el cuerpo. Por último, el punto (2b) es directa consecuencia del Ejercicio 5.3.

Juntando todos los resultados obtenidos se demuestra que para toda Función Primitiva Recursiva existe una rutina FOR que calcula lo mismo. □

La dirección contraria de este resultado también vale:

Teorema 5.2 *Si un programa* FOR *implementa una función* $f : \mathbb{N}^k \mapsto \mathbb{N}$*, entonces* f *es primitiva recursiva.*

Demostración. Esta demostración también hay que hacerla por casos: a partir de rutinas que sólo devuelven un valor constante (que son claramente primitivas recursivas), se van construyendo rutinas más complejas que contienen if-then-else y bucles. Las llamadas a rutina no hace falta tenerlas en cuenta por los Hechos 5.2 y 5.3. El Ejercicio 5.4 indica cómo se puede demostrar el caso más complejo. □

Hecho 5.4 *Programas* FOR *y Funciones Primitivas Recursivas implementan la misma clase de funciones sobre los números naturales.*

Este resultado implica que un programa o rutina FOR sólo calcula funciones totales, es decir, termina siempre. Por lo tanto, el lenguaje FOR no puede ser equivalente a los lenguajes que se conocen que, como se sabe, pueden definir programas que no terminan.

5.2 El lenguaje WHILE

Este lenguaje es muy parecido al lenguaje FOR; la única diferencia es que, además del bucle **for**, también cuenta con el bucle **while**.

$$
\begin{array}{rcl}
\mathcal{E} & ::= & \mathcal{A} \mid \mathcal{B} \\
\mathcal{A} & ::= & n \mid \mathcal{X} \mid \mathcal{A}+\mathcal{A} \mid \mathcal{A}\times\mathcal{A} \mid \mathcal{A}-\mathcal{A} \mid \mathcal{A}/\mathcal{A} \mid \mathcal{A}\%\mathcal{A} \mid \mathcal{X}(\mathcal{E},..,\mathcal{E}) \\
\mathcal{B} & ::= & \textsf{true} \mid \textsf{false} \mid \mathcal{X} \mid \mathcal{A}=\mathcal{A} \mid \mathcal{A}<\mathcal{A} \mid \neg\mathcal{B} \mid \mathcal{B}\vee\mathcal{B} \mid \mathcal{B}\wedge\mathcal{B} \\
\mathcal{C} & ::= & \textsf{skip} \mid \mathcal{X} := \mathcal{E} \mid \mathcal{C};\mathcal{C} \mid \{\mathcal{C}\} \mid \textsf{if } \mathcal{B} \textsf{ then } \mathcal{C} \; [\textsf{else } \mathcal{C}] \\
 & \mid & \textsf{for } \mathcal{X}=\mathcal{A} \textsf{ to } \mathcal{A} \textsf{ do } \mathcal{C} \mid \textsf{while } \mathcal{B} \textsf{ do } \mathcal{C} \mid \textsf{return } \mathcal{E} \\
\mathcal{F} & ::= & \textsf{proc } \mathcal{X}(\mathcal{X}:\mathcal{T},..,\mathcal{X}:\mathcal{T}) \; [\textsf{returns } \mathcal{T}] \; \{[\mathcal{V}] \; \mathcal{C}\} \\
\mathcal{X} & ::= & x_1 \mid x_2 \mid ... \mid f \mid g \mid ... \\
\mathcal{V} & ::= & \textsf{var } \mathcal{X}:\mathcal{T},..,\mathcal{X}:\mathcal{T}; \\
\mathcal{T} & ::= & \textsf{int} \mid \textsf{bool} \\
\mathcal{P} & ::= & \mathcal{F}..\mathcal{F} \; \mathcal{V} \; \mathcal{C}
\end{array}
$$

Es fácil observar que los programas WHILE pueden calcular más cosas que las Funciones Primitivas Recursivas, porque

- los programas FOR son equivalentes a las FPRs;
- los programas WHILE son al menos los mismos que los programas FOR (FOR es un subconjunto de WHILE);
- sólo los programas WHILE pueden implementar funciones que no terminan, ya que el bucle **while** puede no terminar.

Por lo tanto hay funciones que se pueden implementar con programas WHILE pero no con programas FOR ni con Funciones Primitivas Recursivas.

5.2.1 Lenguaje WHILE vs. Funciones μ-Recursivas

Nos preguntamos entonces si los programas WHILE calculan más o menos funciones que las funciones μ-recursivas.

Teorema 5.3 *Dada una función (parcial) de k argumentos $f : \mathbb{N}^k \mapsto \mathbb{N}$ que se puede representar como Función μ-Recursiva, existe una rutina escrita en el lenguaje* WHILE *que recibe k inputs $n_1..n_k$ y devuelve $f(n_1..n_k)$.*

Demostración. La demostración es fácil dado el Teorema 5.1 y el Ejercicio 5.7, que tiene en cuenta el único caso nuevo (la minimización). □

También vale lo contrario: todo programa WHILE se puede representar con una función μ-recursiva. El Ejercicio 5.8 introduce la idea fundamental.

Teorema 5.4 *Si una función (parcial) de k argumentos $f : \mathbb{N}^k \mapsto \mathbb{N}$ se puede implementar como rutina* funF *en* WHILE*, entonces es μ-recursiva.*

Demostración. El Teorema 5.2 y el Ejercicio 5.8 dan una idea de cómo se puede demostrar este hecho. □

El resultado más importante de este capítulo es que todo lo que podemos hacer con WHILE, lo podemos hacer con una μ-Recursiva, y vice versa.

Hecho 5.5 *Una función (posiblemente parcial) de k argumentos $f : \mathbb{N}^k \mapsto \mathbb{N}$ se puede implementar como rutina* funF *en* WHILE*, si y sólo si es μ-recursiva.*

5.2.2 Lenguaje WHILE vs. Máquinas de Turing

Con respecto a la relación entre el Lenguaje WHILE y las Máquinas de Turing, dos ejercicios (cuya solución se encuentra en el Apéndice A) proporcionan los elementos fundamentales de la discusión. El Ejercicio 5.11 da una idea de cómo para todo programa WHILE se puede definir una MT que lo simula. El Ejercicio 5.13 es una verdadera demostración de que cualquier Máquina de Turing se puede "compilar" a un programa WHILE.

La conclusión es que, al igual que sucede con WHILE y funciones μ-recursivas parciales, los programas WHILE y las Máquinas de Turing calculan las mismas funciones.

Teorema 5.5 *Las funciones parciales de k argumentos $f : \mathbb{N}^k \mapsto \mathbb{N}$ que se pueden implementar como rutinas* funF *en* WHILE *son las mismas que se pueden calcular por medio de Máquinas de Turing.*

Demostración. Una dirección de la demostración viene del Ejercicio 5.13, lo otra es una generalización del Ejercicio 5.11. □

Hecho 5.6 *Los programas escritos en el Lenguaje* WHILE *y las Máquinas de Turing calculan la misma clase de funciones calculables.*

Teorema 5.6 *Las funciones parciales de k argumentos $f : \mathbb{N}^k \mapsto \mathbb{N}$ que se pueden calcular con funciones μ-Recursivas parciales son las mismas que se pueden calcular por medio de Máquinas de Turing.*

Demostración. Este teorema no es otra cosa que la aplicación de la propiedad transitiva sobre los Teoremas 5.5 y 5.6: si las μ-recursivas parciales calculan lo mismo que los programas WHILE, y estos calculan lo mismo que las Máquinas de Turing, entonces las μ-recursivas parciales calculan lo mismo que las MTs. □

Hecho 5.7 *Las Funciones μ-Recursivas y las Máquinas de Turing calculan la misma clase de funciones calculables.*

Una última, importante observación es que, a pesar de su sencillez, el Lenguaje WHILE calcula todo lo que se puede calcular con un lenguaje de programación: características más avanzadas como tipos de datos complejos, objetos, gestión dinámica de la memoria, recursión, concurrencia, etc. no permiten *ninguna función* a la clase de funciones calculadas. Otro paradigmas

de programación como el funcional o el lógico tampoco se escapan de estas limitaciones.

Hecho 5.8 *Ningún lenguaje de programación calcula ninguna función que el Lenguaje* WHILE *no pueda calcular.*

5.3 Resumen

Resumen 5.1 *En un lenguaje imperativo, los algoritmos calculados por programas recursivos se pueden calcular también con programas sin recursión.*

Resumen 5.2 *El lenguaje* FOR *y las Funciones Primitivas Recursivas calculan exactamente la misma clase de funciones calculables, todas ellas funciones totales.*

Resumen 5.3 *Las Máquinas de Turing, las Funciones μ-Recursivas y el lenguaje* WHILE *calculan exactamente la misma clase de funciones calculables, y estrictamente más que el lenguaje* FOR *y las Funciones Primitivas Recursivas.*

Resumen 5.4 *Aunque cuente con mecanismos más avanzados desde el punto de vista de la programación, ningún lenguaje de programación calcula ninguna función que el Lenguaje* WHILE *no pueda calcular.*

5.4 Ejercicios

Ejercicio 5.1 *Implementar la función "factorial" con una rutina escrita en el lenguaje* FOR*, teniendo en cuenta que no se puede usar la recursión.*

Ejercicio 5.2 *Implementar una rutina escrita en el lenguaje* FOR *que calcule, dado un input n el n-ésimo término de la serie de Fibonacci, teniendo en cuenta que no se puede usar la recursión.*

Ejercicio 5.3 *Sean* funG *y* funH *funciones escritas en el lenguaje* FOR *que implementan dos funciones recursivas g con $k-1$ argumentos y h con $k+1$ argumentos. A partir de* funG *y* funH*, implementar una función* funF *que calcule f definida como*

$$\begin{cases} f(0, n_2, .., n_k) &= g(n_2, .., n_k) \\ f(n_1 + 1, n_2, .., n_k) &= h(n_1, f(n_1, n_2, .., n_k), n_2, .., n_k) \end{cases}$$

Ejercicio 5.4 *Sea g una función primitiva recursiva de k argumentos que calcula lo mismo que una función* funG *escrita en el lenguaje* FOR. *Definir una FPR f de k argumentos que calcule lo mismo que*

```
1 proc funF(x1:int,..,xk:int) returns int {
2   var x:int;
3   var y:int;
4   y := 0;
5   for x=1 to x1 do y := y + funG(x,x2,..,xk);
6   return y;
7 }
```

Ejercicio 5.5 *Dada la siguiente rutina*

```
1 proc fpr1(x:int) returns int {
2   var z:int;
3   var y:int;
4   z := 0;
5   for y=0 to x do z := z+y+1;
6   return z+2;
7 }
```

escribir la función que calcula como Función Primitiva Recursiva.

Ejercicio 5.6 *Dada la siguiente rutina*

```
1 proc fpr2(x:int) returns int {
2   var y:int;
3   var w:int;
4   if (x%2 = 0) then { w := 1; } else { w := -1 };
5   for y=1 to x do w := w*y*2;
6   return w;
7 }
```

escribir la función que calcula como Función Primitiva Recursiva.

Ejercicio 5.7 *Sea* funG *una rutina implementada en el lenguaje* WHILE *que calcula un predicado μ-recursivo g con $k+1$ argumentos. A partir de* funG, *implementar una rutina* funF *que calcule f de k argumentos definida como*

$$f(n_1,..,n_k) = \begin{cases} \textit{el mínimo } m \textit{ tal que } g(n_1,..,n_k,m) = 1 & \textit{si existe} \\ 0 & \textit{si no existe} \end{cases}$$

Ejercicio 5.8 *Sea g un predicado μ-recursivo de $k+1$ argumentos que calcula lo mismo que una función* funG *escrita en el lenguaje* WHILE. *Sea h una función μ-recursiva de $k+1$ argumentos que calcula lo mismo que una función* funH *escrita en el lenguaje* WHILE. *Definir una función μ-recursiva f de k argumentos que calcule lo mismo que*

```
1 proc funF(x1:int,...,xk:int) returns int {
2   var x:int;
3   var y:int;
4   x := 0;
5   y := 0;
6   while ¬(funG(y,x1,...,xk)=1) do {
7     x := x + funH(y,x1,...,xk);
8     y := y + 1;
9   }
10  return x;
11 }
```

Ejercicio 5.9 *Implementar una rutina en el lenguaje* WHILE *que calcule la función de Ackermann (Ejemplo 4.16, teniendo en cuenta de que no se puede usar la recursión.*

Ejercicio 5.10 *Dada la siguiente rutina*

```
1 proc mu(x:int) {
2   var y:int;
3   y := 100;
4   while (y>0) {
5     if (x>10) then {
6       y := y−1;
7       x := x+1;
8     } else {
9       y := y+1;
10      x := x−1;
11    }
12  }
13  return y;
14 }
```

escrita en el lenguaje WHILE, *definir la función que calcula como Función μ-Recursiva.*

Ejercicio 5.11 *Sean M_g y M_h unas MTs que calculan lo mismo que las rutinas* funG *y* funH *de $k+1$ argumentos escritas en el lenguaje* WHILE. *Definir en forma de diagrama una MT M_f que calcule*

```
1 proc funF(x1:int,..,xk:int) returns int {
2   var x:int;
3   var y:int;
4   x := 0;
5   y := 0;
6   while (¬(funG(x1,..,xk,y)=1)) do {
7     x := x + funH(x1,..,xk,y);
8     y := y + 1;
9   }
10  return x
11 }
```

Ejercicio 5.12 *Definir en forma de diagrama una Máquina de Turing que calcule lo mismo que la siguiente rutina:*

```
1 proc p(x:int) returns int {
2   var y:int;
3   y := 0;
4   while (y<x) do {
5     y := y*y;
6   }
7   return y;
8 }
```

Ejercicio 5.13 *Sea M una Máquina de Turing que calcula una función f de k argumentos. Implementar un programa* WHILE *que calcula lo mismo que M simulando cada uno de sus pasos.*

Parte II

Tesis de Church y problemas indecidibles

Capítulo 6

La Tesis de Church

El capítulo 5 indica que los programas WHILE, las funciones μ-recursivas y las Máquinas de Turing *tienen la misma expresividad*: calculan la misma clase de funciones y deciden la misma clase de lenguajes. A nivel práctico, estos distintos formalismos de computación presentan características diferentes:

- Máquinas de Turing: son la base para la descripción de las *máquinas computadoras* en general, y para el estudio de la complejidad; su ventaja es la **intuición**.

- Funciones μ-Recursivas: junto con el λ-*Cálculo*, son la base para la *programación funcional* y la *semántica denotacional*; su ventaja es la **claridad**.

- Lenguaje WHILE: son la base para la *programación imperativa* y la *semántica operacional*; su ventaja es la **familiaridad**.

Existen muchas variantes de cada uno de estos formalismos. Por un lado, hay MTs con multiples cintas, con cintas bidimensionales, no deterministas, con sólo dos símbolos, con sólo dos estados, etc. Por otro lado, se conocen lenguajes orientados a objetos, concurrentes, o con todo tipo de mecanismos de programación avanzados.

Además, existen otros formalismos que también describen la misma idea de computación y *tienen la misma expresividad*: λ-Cálculo, Gramáticas Formales, Máquinas de Post, Autómatas Celulares, Computadores Cuánticos, hasta el

Juego de la Vida (que es un ejemplo de autómata celular) diseñado por John Conway[1].

6.1 Tesis de Church-Turing; Turing-equivalencia

¿Por qué ciertas funciones matemáticas son calculables y otras no? ¿Por qué todos estos formalismos tienen la misma expresividad? No se puede responder definitivamente a estas preguntas, y desde luego este libro no pretende hacerlo. Sin embargo, el siguiente ejemplo puede ayudar a entender mejor este tema reconduciéndolo a algo más familiar para la mayoría de las personas, por lo menos las que tienen alguna formación científica.

Ejemplo 6.1 *Las teorías físicas desarrolladas en el siglo XX, especialmente la Teoría de la Relatividad, se apoyan en un resultado fundamental:*

No hay nada en el universo que se pueda mover
más rápido que la velocidad de la luz.

Aunque aparentemente muy lejana de nuestro día a día, esta ley determina nuestra idea del universo y lo que se puede hacer en él: transmitir todo tipo de señales, observar estrellas y galaxias, viajar en el espacio, etc. La investigación en física alcanza de vez en cuando nuevos descubrimientos que ponen en cuestión la veracidad de este límite absoluto (existencia de los agujeros negros como singularidades, desgarros en el tejido espacio-tiempo, universos paralelos, etc.), y parecen plantear la posibilidad de transmitir señales o realizar desplazamientos sin tener en cuenta este límite de velocidad.

Sin embargo, no hay evidencia de que esto se pueda efectivamente conseguir, y esta frontera de aproximadamente 300000 Kilómetros al segundo parece ser infranqueable: por un lado, emplear cantidades cada vez más grandes de energía para accelerar partículas con masa sólo llega cada vez más cerca de la velocidad de la luz, sin alcanzarla nunca. Por otro lado, la radiación electromagnética siempre se propaga con esa misma velocidad, sin importar la frecuencia/energía de la onda (rayos γ, rayos X, ultravioleta, luz, infrarrojo, microondas, radiofrecuencia).

De manera análoga, en la Teoría de la Computabilidad existe un límite que parece igualmente infranqueable:

[1] John Conway's Game of Life, `http://www.bitstorm.org/gameoflife/`

nada se puede calcular o decidir que no se pueda calcular con una Máquina de Turing, o un programa WHILE*, o una función μ-Recursiva, u otro de los formalismos conocidos que tienen la misma expresividad.*

Esta afirmación se llama *Tesis de Church*, y se puede presentar en distintas versiones equivalentes: por ejemplo, la *Tesis de Church-Turing* es la versión más conocida, y sólo se refiere a las MTs: *Cualquier función que se pueda calcular es Turing-computable, es decir, existe una MT que la calcula.* La energía o la frecuencia de onda del ejemplo anterior corresponden con las mejorías que se pueden aplicar a una MT: número de cintas, número de símbolos, número de estados, no-determinismo, etc. Está claro que, en el caso de la Tesis de Church, lo que se mantiene insuperable no es una velocidad (la eficiencia de una Máquina de Turing se puede aumentar, y muchísimo, mejorando su *hardware/software*), sino *la clase de funciones que se pueden calcular o lenguajes que se pueden decidir.*

Hecho 6.1 *Según la Tesis de Church, toda función efectivamente calculable puede ser calculada por una MT, una Función μ-Recursiva o un programa* WHILE*.*

La elección de las Máquinas de Turing como formalismo "por excelencia" viene de reconocerlas como el formalismo de computación que mejor describe la idea misma de *computación.* Por Ejemplo, Stephen Kleene escribió "la computabilidad según Turing es intrínsecamente convincente" pero "la λ-definibilidad no es intrínsecamente convincente" y "la recursión general todavía menos (su autor Gödel tampoco estaba convencido)"[2]. No fue hasta 1963 que Kurt Gödel se convenció de que sólo las MT son posibles candidatas para representar el concepto de "efectivamente calculable". Esto es un ejemplo de que conceptos que para nosotros son bastante asentados tardaron años o incluso décadas para que hubiese un consenso sobre ellos por parte de la comunidad científica.

Hecho 6.2 *Las MTs se consideran como el formalismo que mejor describe la idea de efectiva computabilidad.*

[2]"Turing's computability is intrinsically persuasive"; but "λ-definability is not intrinsically persuasive" and "general recursiveness scarcely so (its author Gödel being at the time not at all persuaded)". "Turing's computability" se refiere a la calculabilidad con Máquinas de Turing; "λ-definability" se refiere a la calculabilidad con el λ-cálculo; "general recursiveness" se refiere a la calculabilidad con funciones μ-Recursivas. Esta cita es de 1981 pero refleja opiniones anteriores, de la década de 1930.

6.2 La fuerza de la Tesis

La Tesis de Church no es un teorema con demostración. Más bien, es una afirmación sin demostración matemática, basada en la observación, algo parecido a una *ley de la física*. Sin embargo, es aceptada casi universalmente por la comunidad científica. Cutland [6, Sección 3.7] indica dos de las razones principales por las que la Tesis goza de semejante grado de aceptación entre los matemáticos e informáticos de las últimas décadas.

- A lo largo de las últimas decadas muchos y diversos formalismos, introducidos y estudiados de forma independiente, se demostraron calcular la misma clase de funciones.
- No se ha encontrado hasta ahora ninguna función que se pueda aceptar informalmente como calculable y que no se pueda calcular por medio de una Máquina de Turing.

Hecho 6.3 *Hasta ahora nunca se ha encontrado una función que se pueda calcular efectivamente con algún formalismo sin que una MT pueda calcularla.*

6.3 Historia de la tesis de Church

Esta sección resume los pasos principales que han llevado a la afirmación de la Tesis de Church en su forma actual.

En 1889, y basándose en un trabajo de Julius Dedekind, Giuseppe Peano enuncia sus *axiomas* de la aritmética. Se trata de nueve sencillos axiomas que enuncian propiedades de los números naturales como "todo número natural tiene un sucesor" o el *principio de inducción*: "Si el 0 pertenece a un conjunto, y dado un número natural cualquiera, el sucesor de ese número también pertenece a ese conjunto, entonces todos los números naturales pertenecen a ese conjunto".

Dentro de la axiomatización de la artitmética, David Hilbert, el matemático más conocido en ese momento, enuncia en París en el año 1900 sus famosos *problemas*: 23 preguntas que, según él, iban a determinar la investigación en matemáticas a lo largo del siglo XX. Entre ellos, el segundo y el décimo problema introducen el *Entscheidungsproblem* ("*problema de la decisión*"). En particular, el décimo problema se pregunta acerca de la solubilidad (bajo cierta condiciones) de una *ecuación diofántica* por medio de *un número finito de operaciones*. Se trata de una cuestión que tiene mucho que ver con la idea

de computación, porque se plantea como una tarea de la que no sólo importa el resultado final, sino también su viabilidad en un tiempo finito. Más en general, el problema de la decisón se refiere a decidir en un número finito de operaciones si una fórmula se puede demostrar dentro de un sistema lógico.

La respuesta negativa a esta pregunta fu dada por Kurt Gödel en 1931 con sus *teoremas de incompletitud*: hay fórmulas verdaderas que no se puede demostrar dentro cierto sistema formal (por ejemplo, la lógica), y tampoco se puede demostrar la *consistencia* (corrección) de un sistema formal dentro de ese mismo sistema. Esto supone una respuesta negativa para el problema de la decisión planteado por Hilbert y, en general, un duro golpe para los que, al igual que el alemán, pensaban encontrar un método sistemático para demostrar todos los teoremas de las matemáticas.

Las décadas de 1920 y 1930 vieron la definición de varios formalismos que pretenden describir satisfactoriamente la idea de computación, como las Funciones Primitivas Recursivas y el λ-cálculo. El trabajo de Ackermann en 1927-1928 identifica funciones "sencillas" (y totales) que no son primitivas recursivas, así que este formalismo ya no se considera candidato a expresar la idea fundamental de computación efectiva, y es descartado en favor (Gödel, 1934) de la "recursión general" (algo parecido a la μ-Recursión).

1936 es el año en que aparece la la versión de Church de la Tesis: Alonzo Church define "efectivamente calculable" como algo que la recursión general y el λ-cálculo hacen; no todos están de acuerdo. Como comentado antes, y según la reflexión de Emil Post en 1936, la idea de "computación efectiva" de Church es derivada de un razonamiento inductivo, así que se puede expresar como una "ley de la naturaleza" y no como una definición. En ese mismo 1936 Alan Turing propone una máquinas que calculan como modelo de un hombre que calcula. Tres años más tarde, el mismo Turing afirma que "Computación efectiva" es lo mismo que "computation con MTs" (Tesis de Church-Turing).

1939 es también el año de importantes resultados de equivalencia entre formalismos: John Rosser demuestra la equivalencia entre recursión, λ-cálculo y computación con Máquinas de Turing, y esto da más fuerza a la idea que exista un límite a lo calculable. Los años siguientes dieron lugar a discusiones sobre el formalismo a elegir como referencia para todos los demás, y como definición principal de "efectivamente calculable". La cosa siguó hasta que la mayoría de los matemáticos en cuestión se convencieron (Gödel terminó haciéndolo en 1963, Kleene se convenció mucho antes) de que las Máquinas de Turing representan satisfactoriamente esta idea.

La última definicón relevante de "Computación por medio de máquinas" la dio Robin Gandy en 1980 como proceso discreto, determinista y limitado en

su alcance por la velocidad de la luz ("efecto local"). Esta nueva formulación no invalida la Tesis de Church.

6.4 Desafíos a la Tesis de Church

A lo largo de los años, muchos matemáticos, lógicas e informáticos han buscado una evidencia de que la Tesis de Church es, en efecto, falsa: para demostrarlo sería suficiente encontrar un formalismo de computación que calcular funciones que una MT no puede calcular. A primera vista, podría parecer una tarea relativamente sencilla: las Máquinas de Turing parecen ser tan limitadas que mejorar sus resultados se presenta como algo trivial. Sin embargo, no es así. Esta sección no pretende cubrir exhaustivamente este tema, sino limitarse a plantear algunos aspectos de la discusión, relativos al ámbito más familiar para un informático: los lenguajes de programación.

La intuición dice que los lenguajes de programación actuales son mucho más complejos que FOR o WHILE, y esto hace surgir la pregunta de si las funciones calculadas por Java, C u otro lenguaje moderno siguen siendo las mismas. En concreto, surgen preguntas como:

1. Hasta ahora se ha hablado de funciones sobre números naturales, pero ¿qué ocurre si se usan *tipos* y los varios *formatos de n'umeros* y *datos abstractos*?

2. ¿Cómo se puede expresar la programación orientada a *objetos* en WHILE o con una Función μ-Recursiva?

3. ¿Se puede modelizar con una MT un concepto tan potente como la *concurrencia*?

4. Los programas vistos hasta ahora sólo reciben un input al principio y producen un output al final de la computación, pero ¿cómo se relaciona esto con los mecanismos de *input-output* de los programas reales?

A primera vista, estas características añaden capacidades que el lenguaje WHILE no tiene a la hora de calcular funciones o decidir lenguajes. Sin embargo, no es así: los siguientes párrafos contestan una a una a estas preguntas.

Tipos de datos complejos. A día de hoy todo lo que se genera durante una computación se almacena como secuencias de bits que, al fin y al cabo, son representaciones binarias de números naturales. Por lo tanto, todo tipo de dato se puede codificar fácilmente como número natural.

Programación orientada a objetos. La programación orientada a objetos facilita la organización del código y la programación, pero no hace posible la implementación de nuevos algoritmos; además, los objetos también se almacenan como secuencias de bits. Para convencerse de este hecho basta observar que un programa a objetos es convertido por el compilador en un programa puramente imperativo sin perder ninguna información relevante, por lo que la ventaja que proporcionan los objetos no puede resultar en un agrandamiento de la clase de funciones calculadas.

Concurrencia. Una discusión parecida vale para la concurrencia: se trata de una herramienta de programación muy potente y útil, pero no añade nada a la clase de algoritmos que se pueden implementar. De hecho, un programa concurrente que realmente se ejecuta en muchos procesadores también se puede ejecutar en una máquina con procesador único.

Programas interactivos. Quizás lo más difícil sea aceptar que un programa interactivo no calcula nada más (desde el punto de vista de la Teoría de la Computabilidad) que uno no interactivo. En los últimos años del Siglo XX hubo una discusión sobre si los programas interactivos son realmente más expresivos que los algoritmos "clásicos" que dado un input calculan un output. El lector interesado puede leer el interesante punto vista de Peter Wegner [22, 23], que afirma que los programas interactivos vulneran la mismísima Tesis de Church, y la respuesta de otros investigadores que defienden la Tesis [17]. El resto de esta sección resume informalmente la discusión. En general, se puede pensar que un programa interactivo P recibe durante su ejecución una serie $I_1..I_{n_i}$ de inputs y emita una serie $O_1..O_{n_o}$ de outputs. Entonces hay dos casos:

- Si todo input I_j es independiente de cualquiera de los outputs O_j que P ha generado antes de leer I_j, entonces no se trata de un programa propiamente interactivo: P recibe el input poco a poco, pero para el resultado de su computación es igual que si recibiese todo el input a la vez antes de empezar.

- En cambio, si los inputs I_j dependen de algún output anterior, se hablaría realmente de interacciones que afectan el resultado de P. En este caso, se necesita incluir en el modelo de computación una entidad externa que, dados los outputs generados por P hasta el momento, produzca los siguientes inputs.

- Si esta entidad es otro programa (o más programas), entonces se trataría de un programa concurrente. Por lo que se comentó en los anteriores párrafos, la capacidad de cálculo del sistema en su conjunto no es superior a la de una Máquina de Turing.
- Si esta entidad es capaz de calcular cosas que no son calculables, entonces se trataría de un *oráculo*, es decir, una entidad que genera datos sin que se sepa cómo los genera. En este caso, P podría realmente calcular funciones que una MT no puede calcular, pero esto no se debe a sus capacidades, sino a las del oráculo. Si el oráculo es otra máquina computadora, entonces no puede calcular funciones no calculables, por lo que P tampoco lo hará. Si no es una máquina entonces el sistema en su conjunto (P más el oráculo) no entra dentro de ningún formalismo de computación, por lo que la Tesis de Church no se puede aplicar.

Hecho 6.4 *La posibilidad de escribir programas interactivos no ha permitido hasta el momento encontrar funciones que sean contraejemplos de la Tesis de Church, es decir, funciones calculadas por un programa interactivo pero no por una Máquina de Turing.*

6.5 Resumen

Resumen 6.1 *La Tesis de Church afirma que toda función calculable se puede calcular con una Máquina de Turing, una función μ-Recursiva o un programa* WHILE. *No se trata de un teorema con una demostración, sino más bien de una ley de la naturaleza, que hasta ahora no ha sido desmentida.*

Resumen 6.2 *El principal punto de fuerza de la Tesis es que hay muchos formalismos que calculan la misma clase de funciones, pero ninguno de ellos ha podido añadir, hasta ahora, una sola función a esta clase.*

Resumen 6.3 *A lo largo de los años las MTs han sido aceptadas como el formalismo que mejor expresa la idea de computación efectiva.*

Resumen 6.4 *Algunos científicos consideran la posibilidad de crear programas interactivos como algo que invalida la Tesis de Church. Sin embargo, no se ha encontrado ninguna función calculada por un programa interactivo que no se pueda calcular con una MT. En el caso de usar un oráculo, no se puede hablar de efectiva computabilidad.*

Capítulo 7

Cardinalidad

La Tesis de Church dice que las funciones calculables son las que se pueden calcular usando una Máquina de Turing. El objetivo de este capítulo es ver cuántas son las funciones calculables, y cuántas son las funciones no calculables, si es que existe alguna.

7.1 Conjuntos finitos

La *cardinalidad de un conjunto* A es el número de elementos de A, y se indica normalmente con $|A|$. Por ejemplo, $|\{\}| = 0$, $|\{1, 3, 5\}| = 3$ y $|\{negro, rojo\}| = 2$. Si A es un conjunto finito, la idea de cardinalidad es muy intuitiva. Formalizando un poco más, un conjunto A tiene cardinalidad n si y sólo es posible encontrar una *correspondencia biunívoca* entre A y el conjunto $\{1, .., n\}$, es decir, si se puede asignar a cada elemento de A un único elemento de $\{1, .., n\}$, y a cada elemento de $\{1, .., n\}$ un único elemento de A. La intuición que está debajo de este concepto es la de *contar*: si una persona tiene delante cierto número de objetos y quiere saber cuántos son, lo que hará es indicar cada uno diciendo "uno, dos, tres, etc." hasta haber agotado todos los objetos. Lo que acaba de hacer es asignar a cada objeto un número, y a cada número un objeto (porque no repiten ni los números ni los objetos). Si los objetos son 15, habrá establecido una correspondencia biunívoca entre ellos y el conjunto $\{1, .., 15\}$, por lo que la cardinalidad viene a ser 15.

La cardinalidad de conjuntos finitos tiene ciertas características que nos resultan muy intuitivas y familiares.

Hecho 7.1 *Quitando elementos a un conjunto, su cardinalidad disminuye.*

Hecho 7.2 *Añadiendo elementos a un conjunto, su cardinalidad aumenta.*

Hecho 7.3 *El* conjunto potencia *de A, es decir, el conjunto $\{ X \mid X \subseteq A \}$ de los subconjuntos de A, tiene cardinalidad 2^n, donde n es la cardinalidad del conjunto A.*

Este último hecho implica que la cardinalidad del conjunto potencia B de un conjunto no vacío A siempre es mayor que la cardinalidad de A.

Ejemplo 7.1 *Sea $A = \{$'d', 'f', 'p', 't'$\}$ un conjunto de símbolos. Los subconjuntos de A son*

$$\begin{array}{cccc} \{\} & \{\text{'d'}\} & \{\text{'f'}\} & \{\text{'p'}\} \\ \{\text{'t'}\} & \{\text{'d'}, \text{'f'}\} & \{\text{'d'}, \text{'p'}\} & \{\text{'d'}, \text{'t'}\} \\ \{\text{'f'}, \text{'p'}\} & \{\text{'f'}, \text{'t'}\} & \{\text{'p'}, \text{'t'}\} & \{\text{'d'}, \text{'f'}, \text{'p'}\} \\ \{\text{'d'}, \text{'f'}, \text{'t'}\} & \{\text{'d'}, \text{'p'}, \text{'t'}\} & \{\text{'f'}, \text{'p'}, \text{'t'}\} & \{\text{'d'}, \text{'f'}, \text{'p'}, \text{'t'}\} \end{array}$$

así que la cardinalidad del conjunto potencia de A es $2^{|A|} = 2^4 = 16$.

El último hecho que hay que destacar, también intuitivo en el caso de conjuntos finitos, será muy importante a la hora de hablar de conjuntos infinitos.

Hecho 7.4 *Se puede establecer una correspondencia biunívoca entre dos conjuntos si y sólo si tienen la misma cardinalidad.*

Por ejemplo, ambos conjuntos $A = \{negro, rojo\}$ y $B = \{perro, gato\}$ tienen cardinalidad 2 porque se puede establecer unas correspondencias biunívocas con $\{1, 2\}$ que es el "conjunto de referencia" entre los que tienen cardinalidad 2. Concretamente, si *negro* corresponde a 1 y *rojo* corresponde a 2, se da una correspondencia entre A y $\{1, 2\}$; además, si *perro* corresponde a 1 y *gato* corresponde a 2, se da una correspondencia entre B y $\{1, 2\}$. Entonces, también hay una correspondencia biunívoca entre A y B: por ejemplo, *negro* corresponde a *perro* y *rojo* corresponde a *gato*.

7.2 Funciones calculables: primera parte

Ya se ha abordado un tema parecido en el Capítulo 4 (Observación 4.2 en la Sección 4.3): las Funciones Primitivas Recursivas se pueden *enumerar*, es

decir, existe un método para generarlas una tras otra y asignar a cada una un número natural. En otras palabras, se puede establecer una correspondencia biunívoca entre el conjunto $\mathcal{P}$ y los números naturales[1].

Para mostrar que las FPRs se pueden enumerar, la Observación 4.2 se basa en que la definición de una Función Primitiva Recursiva es una cadena *finita* de símbolos, por lo que la enumeración se efectúa produciendo antes las definiciones más cortas, y a continuación añadiendo más símbolos (Observación 4.1). Sin embargo, la misma observación vale para los demás formalismos que hemos visto hasta ahora: funciones μ-recursivas, programas FOR, programas WHILE y, finalmente, Máquinas de Turing se pueden definir todos como cadenas finitas de símbolos. Esto nos lleva a

Hecho 7.5 *Para todo formalismo de computación, se puede enumerar todo lo que se puede definir con él.*

Ejemplo 7.2 *Enumerar los programas Java es conceptualmente muy sencillo: al ser un programa Java una cadena de símbolos ASCII que son 256, lo que se puede hacer es:*

```
1  k := 1;
2  n := 0;
3  mientras (verdadero) {
4    para cada una de las 256^k cadenas de k símbolos {
5      ejecutar el compilador Java sobre ella;
6      si (compila sin dar error) entonces {
7        marcar la cadena como n–ésimo programa;
8        n := n+1;
9      }
10   }
11   k := k+1;
12 }
```

Obviamente, muy pocas cadenas de símbolos corresponden a programas Java correctos, pero tarde o temprano todo programa aparecerá en la secuencia con su número único n.

Las Máquinas de Turing también se pueden enumerar usando un mecanismo parecido. De hecho, lo que se hace normalmente es enumerar las funciones calculables con respecto a las Máquinas de Turing que las calculan: sea M_n

[1]En la observación 4.2 se empieza a contar desde 0, mientras que en la Sección 7.1 se empieza desde 1. Es preciso subrayar que, en el caso de conjuntos infinitos, ambas opciones son equivalentes (Ejemplo 7.3).

la n-ésima MT según cierta enumeración; entonces φ_n^k es la función calculable parcial con k argumentos definida por la MT M_n. La notación φ_n es una forma de decir φ_n^k cuando k está claro (normalmente $k = 1$, es decir, sin perder generalidad, se suele considerar funciones con un argumento).

7.3 Conjuntos infinitos

Algunos de los resultados presentados en la Sección 7.1 dejan de valer si los conjuntos en cuestión son infinitos. Por ejemplo, dado un conjunto infinito A, los conjuntos $A \cup B$ y $A \setminus B$, donde B es un conjunto finito, tienen exactamente la misma cardinalidad que A. Es más: $A \cup B$ tiene la misma cardinalidad que A incluso si B es infinito, con la única condición de que $|B| \leq |A|$, es decir, que B no sea "más grande" que A.

Hecho 7.6 *Si $|B| \leq |A|$, entonces $|A \cup B| = |A|$.*

Hecho 7.7 *Si A es infinito y B es finito, entonces $|A \setminus B| = |A|$.*

Entre otras cosas, estos resultados implican que quitar 100 números del conjunto de los números naturales o añadir 100 número no naturales al conjunto *no cambia* su cardinalidad. Estos hechos parecen contra-intuitivos, pero desde luego no más que éste:

Hecho 7.8 *Si A es infinito entonces $\left|A^k\right| = |A|$ donde $k \geq 1$ y A^k es el conjunto $\{\ (n_1, .., n_k) \mid n_i \in A\ \}$.*

Es decir, la cardinalidad de los números naturales es igual que la de las *parejas* de números naturales, igual que las *triplas* de números naturales, etc. Obviamente, esto no pasa con conjuntos finitos: si A tiene tres elementos a, b y c, las parejas obtenidas con elementos de A son 9: (a, a), (a, b), (a, c), (b, a), (b, b), (b, c), (c, a), (c, b), y (c, c); además, hay 27 triplas, y 81 "tuplas" de 4 elementos, etc.

¿Por qué suceden estas cosas si se manejan conjuntos infinitos? Es evidente que no se puede contar los elementos de un conjunto infinito *hasta el final* en un tiempo finito, así que la intuición no acaba de acompañar a la hora de decir *cuántos elementos* contiene un conjunto. Por lo tanto, la manera de asignar una cardinalidad a un conjunto infinito sólo puede ser buscar correspondencias biunívocas con otros conjuntos:

dos conjuntos A y B tienen la misma cardinalidad si y sólo es posible encontrar una correspondencia biunívoca *entre ellos.*

Esta definicíon es compatible con la intuición si los conjuntos son finitos, pero da resultados sorprendentes en el caso de conjuntos infinitos.

Ejemplo 7.3 *El conjunto* $\mathbb{N}$ *de los números naturales con el 0 tiene la misma cardinalidad que los números naturales sin el 0,* $\mathbb{N}^+$*. De hecho, la correspondencia biúnivoca es la función* $\lambda n.n + 1$ *que asocia el número 0 de* $\mathbb{N}$ *con el número 1 de* $\mathbb{N}^+$*, el número 1 de* $\mathbb{N}$ *con el número 2 de* $\mathbb{N}^+$*, etc.*

$\mathbb{N}$	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
$\mathbb{N}^+$	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	...

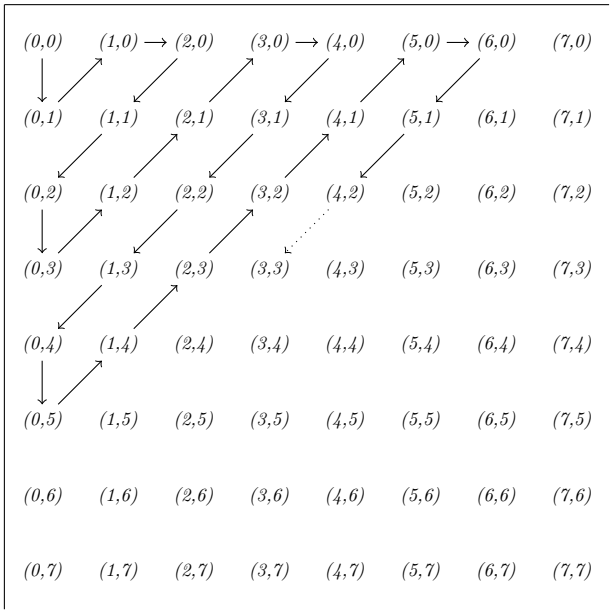
Se nota que las dos enumeraciones cubren todos *los elementos de ambos conjuntos sin dejar ninguno fuera: es exactamente lo que se pide a una correspondencia biunívoca.*

El ejemplo anterior indica que quitar elementos a un conjunto (en este caso, sólo el 0) no tiene por qué cambiar su cardinalidad.

Ejemplo 7.4 $\mathbb{N}$ *tiene la misma cardinalidad que el conjunto* $\mathbb{P}$ *de los números pares. En este caso la correspondencia biúnivoca es* $\lambda n.2n$*:*

$\mathbb{N}$	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
$\mathbb{P}$	*0*	*2*	*4*	*6*	*8*	*10*	*12*	*14*	*16*	*18*	...

Ejemplo 7.5 $\mathbb{N}$ *tiene la misma cardinalidad que el conjunto* $\mathbb{N}^2$ *de las parejas de números naturales. En este caso para encontrar la correspondencia biúnivoca se necesita una forma de enumerar las parejas; se trata de una idea que ya hemos visto anteriormente (Sección 3.4.2). Si los números naturales "caben" en una línea, las parejas de naturales caben en un plano de dos dimensiones, y existe una manera de tocar todos los puntos de un plano infinito sin dejar ninguno fuera:*



La correspondencia biunívoca viene a ser

$\mathbb{N}$	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	...
$\mathbb{N}^2$	*(0,0)*	*(0,1)*	*(1,0)*	*(2,0)*	*(1,1)*	*(0,2)*	*(0,3)*	*(1,2)*	*(2,1)*	*(3,0)*	...

Un razonamiento parecido permite encontrar una correspondencia biunívoca entre $\mathbb{N}$ *y cualquier* $\mathbb{N}^k$ *para* $k \geq 1$*, por lo que todos estos conjuntos tienen la misma cardinalidad, es decir, el mismo número de elementos.*

Un fácil corolario del ejemplo anterior es que el conjunto $\mathbb{Q}$ de los números racionales también tiene la misma cardinalidad de $\mathbb{N}$.

En general, un conjunto se dice *enumerable* si existe una correspondencia biunívoca entre él y $\mathbb{N}$. En otras palabras, un conjunto es enumerable si se encuentra la forma de "contar" sus elementos asignando a cada uno de ellos un número natural partiendo de 0 y sin dejar fuera ningún elemento.

Hecho 7.9 *Los siguientes conjuntos son enumerables:* $\mathbb{N}$, $\mathbb{N}^+$ *(naturales sin* 0*),* $\mathbb{P}$ *(pares),* $\mathbb{N}^k$ *para todo* $k \geq 1$ *(tuplas),* $\mathbb{Q}$ *(números racionales).*

Se podría pensar que todo conjunto infinito es enumerable, es decir, que siempre existe una manera de asignar de forma unívoca un número natural a

todos y cada uno de sus elementos. Sin embargo no es así: el contraejemplo más sencillo es el *conjunto potencia* de $\mathbb{N}$, y el método para demostrarlo es la bien conocida *diagonalización*.

Lema 7.1 *El conjunto potencia* $\wp(\mathbb{N})$ *no es numerable.*

Demostración. Se busca una enumeración de $\wp(\mathbb{N})$. Si ésta existe, se va a poder escribir *todos* los elementos de $\wp(\mathbb{N})$ en una tabla infinita donde cada línea contiene una representación de un sub-conjunto $X \subseteq \mathbb{N}$: en la $(n+1)$-ésima columna, la línea correpondente a X contiene un 1 si $n \in X$, o un 0 si $n \notin X$. Por ejemplo, la siguente tabla contiene en sus primeras cuatro líneas las representaciones de los conjuntos $X_0 = \{0, 2, 4, 6, 8\}$, $X_1 = \{3, 4, 5\}$, $X_2 = \{2, 7\}$, y $X_3 = \{\ n \mid n \geq 4\ \}$, respectivamente.

X_0:	1	0	1	0	1	0	1	0	1	0	...	(todos 0 a partir de aquí)
X_1:	0	0	0	1	1	1	0	0	0	0	...	(todos 0 a partir de aquí)
X_2:	0	0	1	0	0	0	0	1	0	0	...	(todos 0 a partir de aquí)
X_3:	0	0	0	0	1	1	1	1	1	1	...	(todos 1 a partir de aquí)

Si existiera una enumeración completa de $\wp(\mathbb{N})$, todo sub-conjunto de $\mathbb{N}$ aparecería en alguna línea de la tabla. Sin embargo, sea X_n el conjunto representado en la $(n+1)$-ésima línea: el conjunto Y construido sobre la diagonal de la tabla es tal que $Y = \{\ n \mid n \in X_n\ \}$. Este conjunto podría ser igual que algún X_n, pero su complementario $\bar{Y} = \{\ n \mid n \notin X_n\ \}$ *no* puede ser igual a ninguno de los X_n porque difiere de cada X_n por lo menos en el "puesto" n: si $n \in X_n$ entonces $n \notin \bar{Y}$, y si $n \notin X_n$ entonces $n \in \bar{Y}$. En otras palabras, $\bar{Y}$ no aparece en la tabla porque no es representado por ninguna de sus líneas. Por lo tanto, la primera hipótesis que fuera posible enumerar todos los elementos de $\wp(\mathbb{N})$ es falsa, porque siempre hay un $\bar{Y}$ que pertenece a $\wp(\mathbb{N})$ pero no está en la tabla, y esto pasa para *cualquier* intento de enumerar $\wp(\mathbb{N})$. □

Hecho 7.10 *El conjunto* $\wp(\mathbb{N})$ *de los subconjuntos de* $\mathbb{N}$ *no es enumerable.*

Un corolario muy directo es que las funciones de $\mathbb{N}$ a $\{0, 1\}$ tampoco se pueden enumerar. Esto se demuestra al observar que la misma tabla usada en la anterior demostración representa las funciones de números naturales a $\{0, 1\}$.

Hecho 7.11 *Las funciones totales de* $\mathbb{N}$ *a* $\{0, 1\}$ *no son enumerables.*

Finalmente, las funciones de $\mathbb{N}$ a $\mathbb{N}$ son un super-conjunto de las funciones de $\mathbb{N}$ a $\{0,1\}$; por lo tanto, tampoco son enumerables.

Hecho 7.12 *El conjunto de las funciones totales de* $\mathbb{N}$ *a* $\mathbb{N}$ *no es enumerable.*

La cardinalidad de estos conjuntos no es simplemente mayor que la de $\mathbb{N}$: es *infinitamente* mayor. Para dar una idea, se puede pensar que $\mathbb{N}^k$, aparentemente mucho más grande que $\mathbb{N}$, en realidad tiene el mismo número de elementos, mientras $\wp(\mathbb{N})$ tiene más elementos, por lo que tiene que ser *muchísimo*, *infinitamente* más grande que $\mathbb{N}$.

Ejemplo 7.6 *Para darse cuenta de la diferencia entre la cardinalidad de* $\mathbb{N}$ *y su conjunto potencia, se considere el caso de conjuntos finitos: si* $|A| = 4$ *su conjunto potencia tiene 16 elementos, si* $|A| = 6$ *su conjunto potencia tiene 64 elementos, si* $|A| = 8$ *su conjunto potencia tiene 256 elementos, y a medida que el número de elementos de A crece la diferencia con su conjunto potencia se vuelve enorme. Llevando al límite este ejemplo, es posible apreciar mejor que* $\wp(\mathbb{N})$ *es inifitamente más grande que* $\mathbb{N}$.

7.4 Funciones calculables: segunda parte

Los resultados vistos en este capítulo contestan la pregunta "al fin y al cabo, ¿hay algo que no sea calculable?". Y lo hacen de forma sorprendente: la funciones no calculables son infinitamente más que las funciones calculables. De hecho, en la Sección 7.2 quedó claro que las funciones calculable son enumerables[2], mientras que el Hecho 7.12 nos dice que las funciones matemáticas (calculables o no) de naturales a naturales son infinitamente más. Por lo tanto, *casi ninguna* de las funciones de $\mathbb{N}$ a $\mathbb{N}$ es calculable.

Los límites puestos por la Tesis de Church a lo que es calculable se revelan entonces bastante fuertes, ya que dejan fuera la inmensa mayoría de las funciones. Los próximos capítulos presentarán ejemplos de funciones que no son calculable y problemas (lenguajes) que no son decidibles.

[2]Además, al enumerar las Máquinas de Turing (o los programas WHILE) no tenemos en cuenta que dos MTs pueden calcular exactamente la misma función, por lo que podríamos preguntarnos si en realidad las funciones calculables son menos que enumerables; sin embargo, la existencia de infinitas funciones constantes nos asegura que son exactamente enumerables, ya que "enumerable" es la cardinalidad mínima de un conjunto infinito.

7.5 Resumen

Resumen 7.1 *Se puede establacer una correspondencia biunívoca entre dos conjuntos (finitos o infinitos) si y sólo si tienen la misma cardinalidad.*

Resumen 7.2 *Para todo formalismo de computación, se puede enumerar todo lo que se puede definir con él (programas, funciones, algoritmos, etc.). Por lo tanto, las funciones calculable son enumerables.*

Resumen 7.3 *Los siguientes conjuntos son enumerables:* $\mathbb{N}$, $\mathbb{N}^+$, $\mathbb{P}\,\mathbb{N}^k$ *para todo* $k \geq 1$, $\mathbb{Q}$.

Resumen 7.4 *El conjunto potencia* $\wp(\mathbb{N})$ *no es enumerable. Tampoco lo son las funciones totales de* $\mathbb{N}$ *a* $\mathbb{N}$*, que son infinitamente más que los números naturales. Por lo tanto, las funciones calculables son una parte infinitésima de las funciones matemáticas.*

7.6 Ejercicios

Ejercicio 7.1 *Calcular la cardinalidad de* $\wp(\wp(\{a, b, c\}))$.

Ejercicio 7.2 *Demostrar que* $|\mathbb{N}| = |\mathbb{Q}|$.

Ejercicio 7.3 *Demostrar que* $|\mathbb{N}| = \left|\mathbb{N}^3\right| = \left|\mathbb{N}^4\right|$.

Ejercicio 7.4 *Encontrar una correspondencia biunívoca entre* $\mathbb{N}$ *y* $\mathbb{Z}$.

Ejercicio 7.5 *Demostrar que las funciones parciales de* $\mathbb{N}$ *a* $\{0, 1\}$ *no son enumerables, y que tampoco lo son las funciones de* $\mathbb{N}$ *a* $\{0\}$ *(nota: este conjunto tiene un único elemento, pero las funciones son parciales).*

Ejercicio 7.6 *Demostrar que los números reales en el intervalo* $[0, 1)$ *(es decir, incuyendo el 0 pero no el 1) no son enumerables.*

Ejercicio 7.7 *Encontrar una correspondencia biunívoca entre* $(0, 1]$ *(con el 1 pero sin el 0) y los números reales positivos (sin el 0).*

Ejercicio 7.8 *Demostrar que las funciones calculables totales monotonas no-decrecientes de* $\mathbb{N}$ *a* $\mathbb{N}$ *(es decir,* $n_1 < n_2$ *implica* $f(n_1 \leq f(n_2))$ *son infinitas.*

Ejercicio 7.9 *Demostrar que las funciones calculables totales estrictamente crecientes de* $\mathbb{N}$ *a* $\mathbb{N}$ *(es decir,* $n_1 < n_2$ *implica* $f(n_1) < f(n_2))$ *son infinitas.*

Ejercicio 7.10 *Demostrar que las funciones totales estrictamente crecientes de $\mathbb{N}$ a $\mathbb{N}$ (es decir, $n_1 < n_2$ implica $f(n_1) < f(n_2)$) no son enumerables.*

Ejercicio 7.11 *Establecer la cardinalidad del conjunto de funciones parciales de naturales a naturales cuyo dominio tiene un elemento.*

Ejercicio 7.12 *Establecer la cardinalidad del conjunto de funciones parciales de naturales a naturales cuyo dominio tiene como mucho 2 elementos.*

Ejercicio 7.13 *Establecer la cardinalidad del conjunto de funciones parciales de naturales a $\{0, 1\}$ cuyo dominio tiene exactamente 2 elementos.*

Ejercicio 7.14 *Establecer la cardinalidad del conjunto de funciones parciales de naturales a $\{0\}$ cuyo dominio tiene exactamente 2 elementos.*

Ejercicio 7.15 *Establecer la cardinalidad del conjunto de funciones parciales de naturales a naturales cuyo dominio tiene un número finito de elementos.*

Ejercicio 7.16 *Enumerar todos los conjuntos finitos de números naturales mediante una función biyectiva y calculable (es decir, enumerarlos de forma algorítmica).*

Capítulo 8

Problemas indecidibles

Este capítulo mostrará algunos problemas que se han demostrado ser indecidibles en la Teoría de la Computabilidad. En este contexto, las palabras *problema indecidible* pueden significar tanto "función no calculable" como "lenguaje no decidible".

El primer resultado interesante es que para toda función calculable existen infinitas Máquinas de Turing que la calculan. Es decir, estas máquinas hacen exactamente lo mismo pero no son iguales. Dada una M que calcula una función f, se puede imaginar una serie infinita de máquinas $M^1, .., M^k, ..$ que calculan la misma función y difieren de M en que, en lugar de terminar de la misma forma que M, dan cada una un número distinto k de pasos "inútiles" (es decir, que no cambian el contenido de la cinta) al final de la computación: la definición de cada una de estas máquinas es distinta de la de M, pero todas calculan la misma f. En los lenguajes de programación sucede lo mismo: dado un programa, se puede generar una serie infinita de programas equivalentes simplemente añadiendo un número creciente de líneas vaciás en el código: los programas son todos distintos pero obviamente calculan el mismo algoritmo.

Hecho 8.1 *Para toda función calculable existen infinitas Máquinas de Turing que la calculan.*

8.1 La Máquina de Turing universal

Entre todas las Máquinas de Turing existe una máquina especial que se llama *universal*. En realidad, por el Hecho 8.1, existen infinitas MTs universales, y se

puede escoger una cualquiera de ella para desarrollar el resto de este capítulo.

¿Cuál es la característica fundamental de la MT universal? Esta máquina, que llamamos U, recibe como input en la cinta *la descripción "M" de una Máquina de Turing M* y *la descripción de un input w de esta misma M*, y su trabajo es ejecutar M sobre el input. El resultado final de U será el mismo resultado $M(w)$ que se obtendría ejecutando M sobre w, si esta computación termina. Si M calcula la función f, el resultado $U(\text{“}M''\text{”}, w)$ es igual que $f(w)$. Si M no termina para el input w, entonces U tampoco terminará su computación para "M" y w.

En términos informáticos, U viene a ser un *intérprete* de Máquinas de Turing, es decir, un artefacto que ejecuta programas. Por ejemplo, la *Máquina Virtual de Java* (JVM o Java Virtual Machine) es un intérprete de programas escritos en el lenguaje Java bytecode, que a su vez son el resultado de la compilación de programas Java. Un procesador es un intérprete de programas "ejecutables", escritos en lo que se llama *lenguaje de máquina*, que son el resultado de compilar programas escritos, por ejemplo, en C.

Lo interesante de U es que encierra en una única MT toda la potencia de cálculo del formalismo, es decir, es una Máquina de Turing que por sí sola calcula todo lo calculable. Su definición es bastante compleja, y aquí se va a dar simplemente una idea intuitiva. En todo caso, la existencia de intérpretes en el mundo real hace evidente que U realmente se puede definir como Máquina de Turing.

Definición 8.1 *La* Máquina de Turing universal U *se puede definir como una MT con 4 cintas. La Sección 3.4 muestra que máquinas con múltiples cintas calculan exactamente lo mismo que las que sólo tienen una cinta, y este resultado permite definir U como MT con 4 cintas sabiendo que dicha definición se puede transformar en la de una máquina "estándar". La actividad de U consiste de tres fases: (1) fase U_1: inicializar el contenido de todas las cintas; (2) fase U_2: simular reiteradamente los pasos de M hasta llegar a uno de sus estados finales, si esto sucede en un tiempo finito; y (3) fase U_3: si U_2 termina, limpiar las cintas manteniendo sólo el output en la primera.*

- *La cinta C_1 contiene inicialmente el input: la descripción de M y su input w. La fase U_1 mueve w a la cinta C_2, así que esta cinta sólo contiene la representación de M al empezar U_2. Al terminar U_2, si es que termina, C_1 contendrá únicamente el output $M(w)$.*

- *La cinta C_2 contiene, a lo largo de toda la computación de U, una representación de la cinta de M.*

- *La cinta C_3 contiene una representación del estado actual de M, que va cambiando segun su función de transición δ_M almacenada en C_1.*
- *La cinta C_4 contiene un número que simula la actual posición de la cabeza de M en su cinta.*

Dado que es una MT con 4 cintas, la función de transición de U, δ_U es una función de $(K \setminus H) \times \Sigma^4$ a $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})^4$, es decir, U tiene cuatro cabezas, cada una posicionada en su cinta correspondiente, y la acción que se realiza en cada paso es en realidad una acción cuádruple (escribir un símbolo o mover la cabeza en cada una de las cintas) que depende del estado interno y de los cuatro símbolos leídos por las cabezas.

¿Cómo puede U, que, siendo una MT cualquiera, tiene un número finito y fijo de estados y de símbolos, simular todas las MTs, dado que no hay un límite al número máximo de símbolos y estados? Está claro que U no puede usar un estado para simular cada estado de cada M y un símbolo para simular cada símbolo de cada M, porque, por muchos estados y símbolos que tenga U, siempre habrá una M que tenga más estados o más símbolos. La respuesta viene de la observación que, si bien los estados y los símbolos son limitados en U, no lo es la longitud de sus cintas, lo que permite almacenar la información de las configuraciones de M en ellas sin tener que hacerlo en sus estados internos. En su conjunto, una configuración de M en un momento dado de la computación está contenida en C_2 (la cinta), C_4 (la posición de la cabeza) y C_3 (el estado interno).

U tiene que recibir la descripción de M y w con una sintaxis que permita su manipulación. Los números se pueden representar en binario, con cadenas de 0 y 1, pero no sería un problema elegir otra representación. El contenido inicial de C_1 tiene que incluir la descripción completa "M" de M, o por lo menos la parte necesaria para interpretarla: su estado inicial, sus estados finales y su función de transición δ_M. Supongamos que M tenga 50 símbolos y 200 estados internos: si se quiere representar en binario cada uno de sus estados y símbolos, se necesitan 6 bits (porque $2^5 < 52 < 2^6$, donde 52 viene de añadir $\leftarrow$ y $\rightarrow$ al conjunto) para representar los símbolos y 8 bits (porque $2^7 < 200 < 2^8$) para los estados. De este modo U puede simular un número arbitrario de estados y símbolos simplemente usando un número suficiente de bits. Entre los símbolos de U estarán el $\llcorner$, el 0 y el 1, y unos símbolos separadores $|$ y @.

La cinta C_1, después de $\triangleright$ y de un $\llcorner$, empieza con los bits (en este ejemplo, 8) que representan el estado inicial s de M, luego un $\llcorner$, luego el listado de todos los estados finales (cada uno representado por 8 bits) separados por $|$, luego otro $\llcorner$, y finalmente la representación de δ_M. Esta representación es una

serie de cuádruplas, separadas por |, y cada una de ellas contiene, separados por @: un estado (8 bits); un símbolo (6 bits); un estado (8 bits); un símbolo o acción (6 bits). Cada cuádrupla representa exactamente una línea de la tabla que define M. Por ejemplo, el siguiente contenido de C_1

$$\triangleright\llcorner 00000000\llcorner 00001000|01010101\llcorner 00000001@001100@00011111@001011|\ldots$$
$$|00000001@001101@00111111@000000\llcorner \text{“}w\text{”}$$

indica que (1) el estado inicial es $s = q_0$; (2) los estados finales son q_8 y q_{85}; (3) la función δ_M, entre otras cosas, es tal que $\delta_M(q_1, \sigma_{12}) = (q_{31}, \sigma_{11})$ y $\delta(q_1, \sigma_{13}) = (q_{63}, \leftarrow)$ si 000000 es el código reservado para $\leftarrow$. La información representada en C_1 no incluye el número de estados y de símbolos, porque U no lo necesita, pero el número de bits usados para representarlos indica que en M hay como mucho 62 símbolos (porque dos códigos están reservados para $\leftarrow y \rightarrow$) y 256 estados internos.

Después de la representación de δ_M y de otro $\llcorner$ aparece la representación de w, que no es otra cosa que una secuencia de cadenas de 6 bits, cada una indicando un símbolo de M, y separadas por |.

La fase U_1 de inicialización de U consiste en:

- *Mover “w” de C_1 a C_2;*
- *Copiar los bits correspondientes a s de C_1 a C_3;*
- *Escribir el número 1 en C_4, que indica que la posición inicial de la cabeza de M es el primer símbolo después de $\triangleright$; por comodidad, la posición p de la cabeza en la cinta se puede escribir en notación unaria en vez de binaria, como una secuencia de 1.*

Es fácil ver que todos estos pasos son calculables, es decir, existen Máquinas de Turing que los ejecutan y que U puede contener como componentes.

La fase U_2 de simulación de M propiamente dicha se repite las veces que sea necesario, y consiste en:

- *Comprobar si el contenido de C_3 (el estado actual q) es igual a uno de la lista de estados finales; la comparación se hace bit a bit, y si tiene exito con algún elemento de la lista, entonces U da por terminada la simulación de M porque se ha alcanzado un estado final, y pasa a la fase U_3.*
- *Si el paso anterior no ha dado resultado positivo, posicionar la cabeza de C_2 en la posición correspondiente a p, almacenado en C_4. Esto se*

hace contando los símbolos en p (que está representado en unario) y moviéndose cada vez 6 casillas a la derecha en C_2.

- *Buscar q como primer elemento de alguna cuádrupla en la representación de δ_M. Cada vez que se encuentra una cuádrupla que empieza por q, se comprueba si el segundo elemento de la misma cuádrupla es igual que los bits a la derecha de la cabeza en C_2 (hasta el separador). Esto equivale a buscar la línea de la tabla de M que corresponde al estado actual y al símbolo leído; si la comparación del símbolo no tiene éxito, se intenta con otra cuádrupla.*

- *Una vez que se haya encontrado la cuádrupla (q, σ, q', a) correspondiente, se escribe q' en C_3 en lugar de q y se actúa según a:*

  - *si a es un símbolo, entonce se escribe los 6 bits en C_2 en la posición adecuada (donde estaba antes la cabeza correspondiente);*
  - *si es un desplazamiento de la cabeza (nótese que U tiene que saber cuáles son las secuencias de 6 bits que codifican $\leftarrow$ y $\rightarrow$; pueden ser por ejemplo* 000000 *y* 111111*, respectivamente), modifica el valor de p en C_4.*

- *Posicionar las cabezas para preparar el siguiente paso de simulación.*

Todos estos pasos también se pueden reconocer fácilmente como calculables.

La fase U_3 es más sencilla y consta de

- *borrar C_1, C_3 y C_4;*

- *mover el contenido de C_2 (la cinta final de M) a C_1, dejando C_2 vacía.*

Al finalizar la fase U_3, la Máquina de Turing U habrá simulado perfectamente el comportamiento de M sobre el input w. Si la fase U_2 no termina, U tampoco termina, pero en este caso M tampoco terminaría para w.

El siguiente teorema dice que cualquier formalismo que exprese todas las funciones calculables es lo bastante potente para expresar el intérprete de sus programas (la MT U es un ejemplo).

Teorema 8.1 (enumeración o universalidad) *Existe una función calculable parcial de dos argumentos φ_z tal que, para todo x e i, $\varphi_z(i, x) = \varphi_i(x)$*

Demostración. La función φ_z es la función calculada por U. Demostrar este teorema equivale a mostrar que U se puede realmente definir como Máquina de Turing (es decir, que la función que calcula es efectivamente calculable), así que la Definición 8.1 sirve de demostración informal de este teorema. □

Intuitivamente, el Teorema de Enumeración hace innecesaria la presencia de un ser humano que ejecute una MT, porque U proporciona un método automático para hacerlo.

8.2 Funciones calculables e índices

La Observación 4.1 y el hecho que una Máquina de Turing se representa como una secuencia finita de símbolos demuestra que es posible enumerar las Máquinas de Turing de manera que a cada una corresponda uno y un solo número natural. Enumerarlas significa establecer una *correspondencia biunívoca* entre ellas y los números naturales, y esta correspondencia se obtiene a través de una *codificación efectiva*: existe y es calculable la función $\mathcal{C}$ que dada la representación de una MT devuelve un único número natural (*codificación*), y tal que su función inversa $\mathcal{C}^{-1}$, también calculable, dado un número natural devuelve una única representación de una MT (*descodificación*). El número natural $\mathcal{C}(M)$ se llama el *índice* de la MT M, y se dice que la función calculable f tiene índice n si es calculada por la máquina $\mathcal{C}^{-1}(n)$. El siguiente hecho es una reformulación del Hecho 8.1.

Hecho 8.2 *Toda función calculable φ_n tiene infinitos índices, es decir, hay infinitas Máquinas de Turing que la calculan. Por lo tanto existen infinitos números m tales que $\varphi_n = \varphi_m$ (es decir, (1) $\varphi_n(x) = y$ si y sólo si $\varphi_m(x) = y$; (2) $\varphi_n(x) \uparrow$ (no terminación o divergencia) si y sólo si $\varphi_m(x) \uparrow$).*

Este resultado vale, en general, para todo formalismo de computación Turing-equivalente, así que a veces se hablará de índices de "programas" genéricos en lugar de índices de Máquinas de Turing.

Teorema 8.2 (del parámetro, o $s{-}m{-}n$, en su variante $s{-}1{-}1$) *Existe una función* calculable total y biunívoca *s con 2 argumentos, tal que*

$$\forall x, y, z.\ \ \varphi_{s(x,y)}(z) = \varphi_x(y, z)$$

La función s calcula el índice de una función calculable con un argumento z en lugar de dos, que calcula lo mismo que φ_x cuando el primer argumento es cierto valor fijado y.

Demostración. Dado el índice x de una función calculable y dos inputs y y z, s tiene que (1) descodificar x, es decir, encontrar a partir de x el "programa" P que calcula φ_x; (2) sustituir en el "código" de P las referencias al primer parámetro de φ_x con el valor concreto del input y; y (3) codificar el programa P' así obtenido obteniendo su índice. Es fácil ver que s es una función (1) calculable total, porque descodificación, modificación del código y codificación son todas funciones calculables totales; e (2) inyectiva, porque partiendo de índices e inputs distintos se obtienen programas distintos. Tal y como se define aquí, s no tiene por qué se sobreyectiva, pero se puede modificar ligeramente el enunciado del teorema para que la función sea sobreyectiva y, por lo tanto, biunívoca. □

Intuitivamente, el programa o la máquina $P_{s(x,y)}$ sólo opera sobre z, mientras que P_x opera sobre (y, z), así que y es un *parámetro* de P_x. Por ejemplo, sea $\varphi_x(y, z)$ la función $y \cdot f(z)$: entonces, a partir de x y de 2 y usando la s se encuentra de manera efectiva el índice de la MT que calcula la función $2 \cdot f(z)$, es decir, $\varphi_{s(x,2)}$.

Este teorema es importante porque, entre otras cosas, pone las bases de una técnica llamada *evaluación parcial*, que se usa para obtener versiones especializadas y más eficientes de programas cuando se conoce parte de su input. En el ejemplo anterior, $2 \cdot f(z)$ es una especialización de $y \cdot f(z)$ si se conoce el primer argumento.

Ejemplo 8.1 *Sea* p *una rutina con dos argumentos, a la izquierda en la figura. Conocer que el valor del primer argumento es* 3 *permitiría crear una rutina "especializada" que calcula lo mismo en este caso. La especialización más sencilla de* p *sería* p1*, que tiene un argumento y donde el valor* 3 *aparece como constante.*

```
1 proc p(m:int,n:int)
      returns int {
2   if (m<n) then
3     m := m*2;
4   return m;
5 }
```

```
1 proc p1(n:int)
      returns int {
2   m := 3;
3   if (m<n) then
4     m := m*2;
5   return m;
6 }
```

```
1 proc p2(n:int)
      returns int {
2   if (3<n) then
3     return 6;
4   return 3;
5 }
```

Un objetivo de la especialización es obtener programas más eficientes, pero en el caso de p1*, obtenido simplemente asignando el valor 3 a* m*, no se puede decir que esto se consigue (*p1 *es incluso menos eficiente que el original). En general, se necesitan unas técnicas más refinadas de transformación para es-*

pecializar programas de la mejor forma posible. La rutina `p2` *es un ejemplo de especialización que supone una pequeña mejora en la eficiencia porque elimina una operación aritmética. El Teorema del Parámetro garantiza que esta operación se puede llevar a cabo aunque, obviamente, no dice de qué manera. La investigación en el campo de la evaluación parcial intenta encontrar las técnicas de espacialización más eficientes y que dan los mejores resultados.*

El teorema es importante también para el siguiente resultado, que define las condiciones bajo las que un formalismo de computación define todas las funciones calculables.

Teorema 8.3 (expresividad) *Un formalismo de computación es equivalente a las MTs si y sólo si se da una de estas dos cosas:*

- *vale el Teorema de Enumeración (existe la "máquina" universal); o bien*
- *vale el Teorema del Parámetro.*

El Teorema del Parámetro permite demostrar otro importante resultado:

Teorema 8.4 (recursión, Kleene II) *Para toda función calculable total f existe n tal que $\varphi_n = \varphi_{f(n)}$. El índice n se llama* punto fijo *de f.*

Demostración. Se define esta función calculable "diagonal" ψ:

$$\psi(v,z) = \varphi_{d(v)}(z) = \begin{cases} \varphi_{\varphi_v(v)}(z) & \text{si } \varphi_v(v) \text{ converge} \\ \text{indefinida} & \text{en caso contrario} \end{cases}$$

Por el Teorema del Parámetro, se puede elegir d total y biunívoca: de hecho, $d(v)$ es $s(i,v)$ donde s es la función mencionada en el Teorema del Parámetro y ψ es φ_i. Dada la f, se elige un índice v tal que $\varphi_v(x) = f(d(x))$ para todo x. En otras palabras, φ_v es la composición de f y d, y es total porque f y d lo son, así que $\varphi_v(v)$ converge. Por lo tanto, la definición de ψ dice que $\varphi_{d(v)}$ es igual que $\varphi_{\varphi_v(v)}$ porque nunca se elige el segundo caso de la definición de ψ. Cualquier $n = d(v)$ es un punto fijo de f porque

$$\varphi_n = \varphi_{d(v)} = \varphi_{\varphi_v(v)} = \varphi_{f(d(v))} = \varphi_{f(n)}$$

□

n hace que f sea un *transformador de programas*: transforma el "programa" φ_n en $\varphi_{f(n)}$, que calcula exactamente lo mismo (es decir, los programas tienen la misma semántica). Este teorema es la base de la *semántica denotacional*, de los programas que usan el mecanismo de la *recursión*, y de las *funciones de criptografía*, entre otras cosas.

8.3 El Problema de la Parada

La Sección 3.3.1 introdujo el concepto de conjunto *decidible* o *recursivo*: $Y \subseteq \mathbb{N}$ es decidible si existe una MT que termina en un estado de aceptación para todo input $z \in Y$ y termina en un estado de rechazo para todo input $z \notin Y$. Dicho de otra forma, su *función característica* $\mathcal{X}_Y$ definida como

$$\mathcal{X}_Y(z) = \begin{cases} 1 & \text{si } z \in Y \\ 0 & \text{si } z \notin Y \end{cases}$$

es calculable total. Esta segunda definición de conjunto decidible pone de manifiesto la estrecha relación entre conjuntos decidibles y funciones calculables (totales), ya que expresa la decidibilidad de un conjunto a través de la calculabilidad de una función total.

La siguiente función $\phi_{i,s}$ describe lo que calcula una MT M_i con índice i en un número de pasos limitado por s:

$$\phi_{i,s}(x) = \begin{cases} y & \text{si } \varphi_i(x) \text{ es calculado en } n \text{ pasos, con } n \leq s \\ \text{indefinida} & \text{en caso contrario} \end{cases}$$

No es difícil averiguar que es recursivo el conjunto $X_i = \{(x, s) \mid \exists y.\ \phi_{i,s}(x) = y\}$ de los pares (x, s) tales que $M_i(x)$ termina en s pasos como mucho: basta ejecutar M_i sobre el input x limitando el número de pasos a s y, si termina con un resultado y antes de dar todos los s pasos, entonces $(x, s) \in X_i$; si no, $(x, s) \notin X_i$. El punto fundamental en esta limitación es que, incluso si $M_i(x)$ no termina, después de s pasos su computación se aborta y se da una respuesta negativa en lugar de esperar un tiempo infinito.

Además, la Sección 3.3.3 introdujo la definición de conjunto *semidecidible* o *recursivamente enumerable*: Y es semidecidible si se da una de estas condiciones equivalentes:

- existe una MT que termina para un input x si y sólo si $x \in Y$;
- Y es el dominio de una función calculable parcial;
- Y es vacío o es el codominio de una función calculable total.

El lector ya está familiarizado con las primeras dos condiciones, y es fácil observar que son dos maneras distintas de decir lo mismo. Sin embargo, la tercera condición, aún siendo menos intuitiva, es la más parecida a las ideas originales de Turing (y la que más literalmente corresponde a las palabras "recursivamente enumerable"), y su equivalencia a las otras dos no es tan evidente. En lo que sigue se da por supuesto esta equivalencia sin demostrarla.

Dos útiles propiedades de los conjuntos recursivamente enumerables son:

Lema 8.1 *Un conjunto recursivo es recursivamente enumerable. Además, si tanto Y como su complementario son recursivamente enumerables, entonces ambos son recursivos.*

Demostración. La primera afirmación es trivial; la segunda se demuestra observando que, por definición, si tanto Y como $\bar{Y}$ son recursivamente enumerables, entonces existen dos MTs M y $\bar{M}$ que los semideciden, es decir:

- $M(x)$ termina si y sólo si $x \in Y$; y
- $\bar{M}(x)$ termina si y sólo si $x \in \bar{Y}$ si y sólo si $x \notin Y$.

Por hipótesis, para cada input una de las dos máquinas termina. Entonces basta intercalar las computaciones de $M(x)$ y $\bar{M}(x)$ (dando un paso de cada una, alternadamente) hasta que una de las dos termine, y esto dará la respuesta en un tiempo finito sobre la pertenencia de x a Y. □

¿Qué ocurre si se elimina la limitación en el número de pasos dada en $\phi_{i,s}$? Esto significa decidir si una Máquina de Turing termina, es decir, si *existe* un número s tal que la máquina produce un resultado como mucho en s pasos. Un conjunto muy importante en la Teoría de la Computabilidad es

$$K = \{\ x \mid \varphi_x(x) \downarrow\ \}$$

que identifica (los índices de) las funciones calculables que son definidas cuando el input es su propio índice, o, lo que es lo mismo, las MTs que terminan cuando se aplican a su propio índice.

Teorema 8.5 *K es recursivamente enumerable.*

Demostración. Es el dominio de la función parcial

$$F(x) = \begin{cases} x & \text{si } \varphi_x(x) \downarrow \\ \text{indefinida} & \text{en caso contrario} \end{cases}$$

que es calculable porque basta ejecutar la MT M_x sobre el input x. Si $M_x(x)$ termina, entonces también terminará $F(x)$, por lo que x pertenecerá a su dominio. Si $M_x(x)$ no termina, $F(x)$ se quedará esperándola un tiempo infinito y tampoco terminará, por lo que x no pertenecerá al dominio de F. □

Teorema 8.6 *K no es recursivo.*

Demostración. Supongamos que lo sea; entonces su función característica $\mathcal{X}_K$ sería calculable total. Pero entonces la función

$$F(x) = \begin{cases} \varphi_x(x) + 1 & \text{si } \mathcal{X}_K = 1 \\ 0 & \text{si } \mathcal{X}_K = 0 \end{cases}$$

sería también calculable total, es decir, una de las φ (se nota que $\varphi_x(x)$ termina siempre que se ejecuta porque previamente se garantiza que $\mathcal{X}_K = 1$). Pero esto es imposible porque, para todo j, $F \neq \varphi_j$ por diagonalización. De hecho, una de estas dos cosas sucede para todo input k: (1) $\varphi_k(k)$ es indefinida pero $F(k) = 0$; o bien (2) $F(k) = \varphi_k(k) + 1$. En los dos casos hay un input para el que F y φ_k difieren, por lo que F no es calculable. Por lo tanto, K no es recursivo. □

Lo que dice este teorema es que *no existe* un algoritmo para decidir si $x \in K$ o no. Este problema es *insoluble*, aunque es semidecidible, es decir, soluble "a medias". Otra cosa importante es que el complementario $\bar{K}$ de K no es ni siquiera recursivamente enumerable (si no, por el Lema 8.1, ambos serían recursivos, pero K no lo es por el Teorema 8.6). En otras palabras, no sólo hemos visto un ejemplo importante de conjunto no recursivo, sino que también ¡acabamos de darnos cuenta de que existen problemas todavía "más insolubles" que K! No vamos a profundizar en este tema, pero de hecho existen infinitos niveles de insolubilidad de problemas. Es importante volver a subrayar que todos estos resultados no dependen del formalismo de computación utilizado: valen para las Máquinas de Turing, los programas WHILE, las funciones μ-Recursivas parciales, etc., es decir, *no podemos quitarnos de encima el problema simplemente inventando un nuevo lenguaje de programación u otro formalismo Turing-equivalente.*

Hemos llegado a lo que se llama *Problema de la Parada* (*Halting Problem*):

dados un input x y un índice y,
¿termina la MT con índice y si se aplica al input x?

Sea K_0 el conjunto

$$\{\ (x, y) \mid \varphi_y(x) \downarrow\ \}$$

de las parejas (input,índice) tales que la MT con el índice y termina para el input x. Decidir este conjunto significaría contestar la pregunta.

Teorema 8.7 *El conjunto K_0 no es recursivo.*

Demostración. Se da que $x \in K$ si y sólo si $(x, x) \in K_0$. Por lo tanto, si K_0 fuera recursivo también lo sería K, que no lo es por el teorema 8.6. □

La no recursividad de K_0 tiene implicaciones muy importantes: dicho de otra forma, el Teorema 8.7 dice que no existe una MT M_A que, si se le da la representación de otra máquina M_B y un input w, es capaz de decidir *siempre* si M_B termina para el input w. Generalizando, no existe un programa P_A que decida, para todo programa P_B y todo input w, si $P_B(w)$ termina.

El programa P_A que buscamos es un programa que coja otro programa P_B y diga si alguna propiedad de P_B vale o no. En términos informáticos, P_A es un *analizador de programas* y la propiedad que investiga es la terminación para cierto input. Entonces el problema de la parada nos dice que no existe un analizador de terminación que siempre acierte.

Hemos llegado a ver problemas insolubles que tienen relevancia para la informática, como la terminación de un algoritmo. El resto de nuestro recorrido será ver que

- muchos conjuntos interesantes desde el punto de vista matemático no son recursivos; y sobre todo
- casi todas las propiedades interesantes que podemos investigar acerca de un programa son indecidibles.

La demostración del Teorema 8.7 usa un interesante mecanismo: establecer un "si y sólo si" con respecto a otro problema cuyas características son conocidas por resultados que ya se han obtenido. Este procedimiento se llama *reducción* y será uno de los temas del próximo capítulo.

8.4 Resumen

Resumen 8.1 *Para toda función calculable existen infinitas (enumerables) Máquinas de Turing que la calculan. Dicho de otra forma, toda función calculable tiene infinitos índices, correspondientes a las MTs que la calculan.*

Resumen 8.2 *La Máquina de Turing universal U calcula por sí sola cualquier función calculable f, si como input se le da la descripción "M" de una MT M que calcula f y un input w de M tal que $U(\text{“}M''\text{”}, w) = M(w) = f(w)$.*

Resumen 8.3 *Para toda función calculable f con dos argumentos y todo input y es posible calcular, de manera efectiva, el índice de otra función g con un*

argumento que calcula lo mismo que f cuando su primero argumento es y, es decir, tal que $g(z) = f(y, z)$ para todo z.

Resumen 8.4 *El conjunto K de las función que terminan cuando se aplican a su propio índice es recursivamente enumerable pero no recursivo.*

Resumen 8.5 *Existen conjuntos como $\bar{K}$ que no son ni siquiera recursivamente enumerables, y hay un número infinito (enumerable) de niveles de indecidibilidad.*

Resumen 8.6 *No existe ningún programa A que, dado otro programa B y un input w, decide si $B(w)$ termina sin equivocarse nunca.*

8.5 Ejercicios

Ejercicio 8.1 *Definir en forma de diagrama la parte de U que leyendo la posición de la cabeza de M en la cuarta cinta se coloca en la casilla adecuada en la segunda cinta, suponiendo que M tiene 10 símbolos.*

Ejercicio 8.2 *Definir en forma de diagrama la parte de U que dado el estado q representado en C_3 lo busca en la primera cinta, teniendo en cuenta toda la información contenida en C_1 (es decir, sabiendo lo que hay que leer y lo qua hay que ignorar).*

Ejercicio 8.3 *Definir en forma de diagrama la parte de U que, después de encontrar la línea correcta de la función de transición de M, cambia el contenido de C_3 almacenando la representación del nuevo estado.*

Ejercicio 8.4 *Definir en forma de diagrama la parte de U que, después de encontrar la línea correcta de la función de transición de M, ejecuta la acción correspondiente, especificando lo que pasa tanto cuando la acción es escribir un símbolo como cuando se trata de mover la cabeza de M.*

Ejercicio 8.5 *Dada la rutina con dos argumentos*

```
1 proc p(m:int,n:int) returns int {
2   var j:int;
3   j := 1;
4   while (m<10) {
5     n := n+j;
6     j := j+1;
```

```
7      m := m+1; }
8    return n
9  }
```

escribir una versión especializada `p1` *de* `p` *para el caso en que* m *sea igual a 7. ¿Es posible definir* `p1` *sin usar ningún bucle? ¿Cuál es la implementación más pequeña (con menos cantidad de código fuente) de* `p1`*? ¿Cuántas versiones distintas de* `p1` *existen? ¿En qué resultados teóricos se fundamenta esta última respuesta?*

Ejercicio 8.6 *Dada la rutina con dos argumentos*

```
1 proc p(m:int,n:int) returns int {
2   if (n>3) then { if (m+n<8) then return 1; }
3   return 0;
4 }
```

escribir una versión especializada `p1` *de* `p` *para el caso en que* m *sea igual a 6, y una para el caso en que* n *sea igual a* 4. *¿Cuál es la implementación más pequeña (con menos cantidad de código fuente) de* `p1` *en cada uno de los dos casos? ¿Cuántas versiones distintas de* `p1` *existen? ¿En qué resultados teóricos se fundamenta esta última respuesta?*

Ejercicio 8.7 *Dada la rutina con dos argumentos*

```
1 proc p(m:int,n:int) returns int {
2   var i:int;
3   i:= 0;
4   while (i<m) {
5     n := n*2;
6     i := i+1; }
7   return n;
8 }
```

escribir una versión especializada `p1` *de* `p` *para el caso en que* m *sea igual a 3. ¿Cuál es la implementación más pequeña (con menos cantidad de código fuente) de* `p1`*? ¿Cuántas versiones distintas de* `p1` *existen? ¿En qué resultados teóricos se fundamenta esta última respuesta?*

Ejercicio 8.8 *Demostrar que un conjunto o lenguaje es recursivamente enumerable si y sólo si es vacío o es el codominio de una función calculable total.*

Ejercicio 8.9 *La Conjetura de Goldbach dice que todo número par es la suma de dos números primos. Es una conjetura porque* todavía no se ha demostrado.

Decir si es calculable el siguiente predicado:

$$g(n) = \begin{cases} 1 & \textit{si la conjetura de Goldbach es verdad} \\ 0 & \textit{en caso contrario} \end{cases}$$

Ejercicio 8.10 *La función no recursiva del* castor laborioso *o* busy beaver *se define como $\beta : \mathbb{N} \mapsto \mathbb{N}$ y es tal que, para todo número n, $\beta(n)$ es el máximo número m tal que existe una MT con alfabeto $\{\triangleright, \_, a, b\}$ y exactamente n estados internos que, si empieza su ejecución con la cinta vacía, termina con la configuración $(h, \triangleright \underline{\ } a^m)$ (a repetido m veces). Dicho de otra forma, m es el máximo número que puede ser escrito en la cinta por una Máquina de Turing con estas características y con n estados.*

Demostrar que, si f es una función calculable, entonces existe un número k_f tal que $\beta(n + k_f) \geq f(n)$. Pista: k_f es el número de estados internos de la MT M_f que, para un input a^n, termina con $a^{f(n)}$ en la cinta.

Ejercicio 8.11 *Sabemos que la clase de lenguajes recursivamente enumerables no es cerrada bajo complementación. Demostrar que lo es bajo unión e intersección, es decir, que si L_1 y L_2 son R.E., entonces $L_1 \cup L_2$ y $L_1 \cap L_2$ lo son.*

Ejercicio 8.12 *Demostrar que la clase de lenguajes recursivos es cerrada bajo complementación.*

Ejercicio 8.13 *Demostrar que la clase de lenguajes recursivos es cerrada bajo unión e intersección.*

Ejercicio 8.14 *Demostrar que la clase de lenguajes recursivos es cerrada bajo concatenación, es decir, que si L_1 y L_2 son recursivos, entonces L definido como $\{\ xy \mid x \in L_1, y \in L_2\ \}$ también lo es, donde xy es la concatenación de las dos cadenas de símbolos x e y.*

Ejercicio 8.15 *Demostrar que es calculable la función*

$$f(x) = \begin{cases} 1 & \textit{si en el desarrollo decimal de } \pi \textit{ aparecen} \\ & \quad \textit{al menos } x \textit{ sietes consecutivos} \\ 0 & \textit{en caso contrario} \end{cases}$$

Ejercicio 8.16 *Demostrar que no es calculable la función*

$$F(x) = \begin{cases} 1 & \textit{si } \varphi_x(x) \textit{ converge} \\ 0 & \textit{si } \varphi_x(x) \textit{ diverge} \end{cases}$$

Capítulo 9

El Teorema de Rice

Este capítulo da los útimos pasos del recorrido teórico de este libro, antes de discutir las implicaciones prácticas de la Teoría de la Computabilidad.

9.1 Clases de problemas

En este contexto, un *problema* es la pertenencia a un conjunto de números naturales: que un problema sea *resoluble* significa que el conjunto es decidible, es decir, que se puede determinar algorítmicamente si cierto input pertenece o no al conjunto.

Ejemplo 9.1 *Determinar si un número es primo es un problema resoluble porque el conjunto de los números primos es decidible.*

Si el problema no tiene una respuesta "sí o no", sino un valor (dado un input, calcular un output), entonces su resolubilidad equivale a la calculabilidad de la función correspondiente.

Ejemplo 9.2 *El problema de calcular, dado un número n, la suma de los primeros n números naturales se expresa de la forma más natural no como pertenencia a un conjunto, sino como función entre input y output. Es un problema resoluble porque la función $\Sigma_{i=1}^{n} i$ es calculable total.*

Como conjunto de referencia se usan los números naturales en lugar de un lenguaje L entendido como conjunto de cadenas de símbolos de cierto alfabeto, como pide el concepto original de decidibilidad (Sección 3.3.1) porque

un lenguaje de cadenas de símbolos tiene cardinalidad finita o como mucho enumerable, así que siempre existe una codífica de todo elemento x de L a un número $n \in \mathbb{N}$.

Ejemplo 9.3 *Sea L el lenguaje $\{a^n b^n | n \geq 0\}$ de las cadenas sobre el alfabeto $\{a, b\}$ que contienen dos secuencias de a y b de igual longitud. Es fácil en este caso codificar cada $a^n b^n$ en el número (representado en binario) $1^{n+1}0^n$, donde se usa un 1 adicional al principio para que la cadena vacía $a^0 b^0$ también tenga una representación númerica (como 1). Los elementos del conjunto complementario $\bar{L}$ también se pueden representar de la misma forma codificando cada ocurrencia de a con un 1 y cada ocurrencia de b con un 0, y añadiendo un 1 al principio de la representación binaria.*

Una *clase de complejidad* o *clase de problemas* es un conjunto de problemas que tienen características comunes con respecto a su *dificultad*: por ejemplo, los problemas resolubles en tiempo lineal y los problemas resolubles en tiempo cuadrático son dos clases de problemas, y la segunda contiene la primera porque si existe un algoritmo lineal que resuelve un problema también existe otro algoritmo cuadrático lo resuelve. La Teoría de la Computabilidad no se ocupa de la cantidad de recursos (en espacio, como la memoria utilizada, o en tiempo) necesaria para resolver problemas, sino simplemente si un problema se puede o no resolver *en un tiempo finito*. Por esto, la clase de complejidad más pequeña que la teoría maneja es la de los problemas *resolubles en un tiempo finito* o *recursivos*, que corresponden a conjuntos decidibles o funciones calculables (totales). Esta clase incluye, entre otros, los problemas resolubles en tiempo constante, logarítmico, lineal, cuadrático, polinomial, exponencial, etc.

Fuera de esta clase se encuentran los problemas correspondientes a conjuntos semidecidibles o recursivamente enumerables, o a funciones calculables parciales, por los que no siempre existe una solución en tiempo finito. Más adelante se verá que existen problemas todavía más difíciles.

9.2 Reducción entre problemas

Uno de los métodos más efectivos para demostrar si un problema pertenece o no a cierta clase de complejidad es compararlo con otros problemas mejor conocidos y establecer si es más o menos difícil que ellos. Este procedimiento se basa en la idea de *reducción* entre problemas. En esta sección los problemas se representan como conjuntos de números naturales.

Dado un conjunto F de funciones, una *relación de reducción* $\leq_F$ satisface $A \leq_F B$ para dos problemas A y B si existe $f \in F$ tal que, para todo x, $x \in A$ si y sólo si $f(x) \in B$. En otras palabras, la función f *transforma* el problema A en el problema B: si se sabe resolver B, entonces para resolver el problema de la pertenencia de x a A se puede transformar x en $f(x)$ y contestar la pregunta sobre la pertenencia de $f(x)$ a B. Una observación importante es que la transformación f no tiene que ser demasiado difícil con respecto a los problemas A y B, porque si no no valdría la pena la transformación.

Ejemplo 9.4 *Sea A un problema cuya complejidad es cuadrática ($\mathcal{O}(n^2)$), y B un problema lineal ($\mathcal{O}(n)$). Sea f_A un algoritmo cuadrático para resolver A, y f_B un algoritmo lineal para resolver B. A primera vista, transformar un input de f_A en un input de f_B y ejecutar este último es más rápido, porque se pasaría de una complejidad cuadrática a una lineal. Pero esto sólo se da si el coste de la transformación f de A a B no es demasiado alto. De hecho, si f fuera un algoritmo con complejidad cuadrática o incluso $\mathcal{O}(n^3)$, no merecería la pena utilizarlo para transformar un problema en el otro.*

Las siguientes definiciones introducen las herramientas matemáticas para comparar problemas.

Definición 9.1 *Sea F un conjunto de funciones, y sean $\mathcal{D}$ y $\mathcal{E}$ dos clases de problemas (clases de conjuntos de números naturales) con $\mathcal{D} \subseteq \mathcal{E}$. Se dice que la relación de reducción $\leq_F$* clasifica *$\mathcal{D}$ y $\mathcal{E}$ si para tres problemas A, B y C:*

- *$A \leq_F A$ (reflexividad);*
- *$A \leq_F B$ y $B \leq_F C$ implica $A \leq_F C$ (transitividad);*
- *$A \leq_F B$ y $B \in \mathcal{D}$ implica $A \in \mathcal{D}$;*
- *$A \leq_F B$ y $B \in \mathcal{E}$ implica $A \in \mathcal{E}$.*

Se trata de una relación reflexiva y transitiva que, además, *ordena* problemas (por esto se usa un símbolo como $\leq$) porque A no puede no pertenecer a $\mathcal{D}$ si $B \in \mathcal{D}$, y A no puede no pertenecer a $\mathcal{E}$ si $B \in \mathcal{E}$, pero podría pasar, por ejemplo, que $A \in \mathcal{D}$ pero $B \notin \mathcal{D}$.

Intuitivamente, $\leq_F$ *clasifica* $\mathcal{D}$ y $\mathcal{E}$ cuando las funciones de F no son demasiado difíciles, en el sentido del Ejemplo 9.4: las funciones F_3 con complejidad $\mathcal{O}(n^3)$ *no* clasifican los problemas lineales y los cuadráticos porque se puede encontrar dos problemas A y B tales que $A \leq_{F_3} B$ y B es lineal, pero A no lo es (o que B es cuadrático, pero A no lo es).

Definición 9.2 *Si $\leq_F$ clasifica $\mathcal{D}$ y $\mathcal{E}$, entonces para problemas A, B y H:*

- *$A \equiv B$ si $A \leq_F B$ y $B \leq_F A$ (equivalencia entre problemas);*
- *H es $\leq_F$-difícil ($\leq_F$-hard o $\leq_F$-complejo) para $\mathcal{E}$ si para todo $A \in \mathcal{E}$ se da que $A \leq_F H$;*
- *H es $\leq_F$-completo ($\leq_F$-complete) para $\mathcal{E}$ si es $\leq_F$-difícil para $\mathcal{E}$ y además pertenece a $\mathcal{E}$.*

Los problemas completos son los "más difíciles" de su clase de complejidad: todos los problemas de su clase se pueden reducir a ellos.

Hecho 9.1 *Si un problema es $\leq_F$-completo para $\mathcal{E}$ y pertenence a una clase $\mathcal{D} \subseteq \mathcal{E}$, entonces $\mathcal{D}$ y $\mathcal{E}$ coinciden.*

Hecho 9.2 *Si A es $\mathcal{E}$-completo[1], $B \in \mathcal{E}$ y $A \leq_F B$, entonces B es también $\mathcal{E}$-completo.*

El siguiente lemma establece unas condiciones en las que la relación de reducción construida sobre un conjunto de funciones clasifica dos clases de problemas. Es importante notar que, aunque se parezcan bastante, la Definición 9.1 y el Lema 9.1 no son equivalentes: la primera es una definición, es decir, la introducción de un nuevo concepto del que se dan las características deseadeas. El segundo es algo que hay que demostrar, porque (en este caso) dice que algunas condiciones son suficientes para que se cumpla una definición.

Lema 9.1 *Una relación de reducción $\leq_F$ clasifica $\mathcal{D}$ y $\mathcal{E}$ si*

- *la identidad pertenece a F;*
- *si $f \in F$ y $g \in F$, su composición pertenece a F (cierre bajo composición);*
- *si $f \in F$ y $B \in \mathcal{D}$, entonces $\{ x \mid f(x) \in B \} \in \mathcal{D}$;*
- *si $f \in F$ y $B \in \mathcal{E}$, entonces $\{ x \mid f(x) \in B \} \in \mathcal{E}$.*

Demostración. Se obtiene fácilmente observando que cada una de estas condiciones asegura la condición correspondiente en la Definición 9.1. □

Sea *REC* la clase de los problemas recursivos, y *RE* la clase de los problemas recursivamente enumerables. El siguiente teorema dice que la relación de reducción basada en las funciones recursivas clasifica *REC* y *RE*.

[1] $\mathcal{E}$-completo es una manera de decir "$\leq_F$-completo para $\mathcal{E}$" si se puede dejar F implícito.

Teorema 9.1 *La relación $\leq_{rec}$ clasifica REC y RE, donde rec es la clase de las funciones recursivas (calculables totales).*

Demostración. Hay que demostrar que las cuatro condiciones del Lema 9.1 se cumplen para $\leq_{rec}$, REC y RE.

- la primera es obvia porque la identidad es recursiva;
- la segunda también es obvia porque la composición de dos funciones recursivas es recursiva;
- hay que demostrar que si $B \in REC$ y f es una función recursiva, entonces $\{ x \mid f(x) \in B \}$ también pertenece a REC; esto es así porque la función característica de $\{ x \mid f(x) \in B \}$ es $f(\mathcal{X}_B(\_))$, que es calculable total porque f y $\mathcal{X}_B$ son ambas totales, así que el conjunto es recursivo;
- hay que demostrar que si $B \in RE$ y f es una función recursiva, entonces $\{ x \mid f(x) \in B \}$ también pertenece a RE; esto se ve porque la función característica de $\{ x \mid f(x) \in B \}$ es $f(\mathcal{X}_B(\_))$, que es calculable parcial porque $\mathcal{X}_B$ es calculable parcial y f es calculable total, así que el conjunto es recursivamente enumerable.

□

Intuitivamente, este teorema dice que las funciones calculable totales no son "demasiado difíciles" para que tenga sentido transformar REC o RE.

Dado un problema RE-completo (es decir, tal que ninguno de los RE es "más difícil" que él), entonces comparándolo con otros problemas se podrá establecer si estos son o no decidibles. Un problema RE-completo es, por ejemplo, el conjunto K (Sección 8.3).

Hecho 9.3 *El conjunto K, visto como problema, es RE-completo.*

Dados dos problemas A y B pertenecientes a clases de problemas clasificadas por cierta relación de reducción $\leq_F$, decir que "A se reduce a B" (en fórmulas, $A \leq_F B$) significa que A no es más difícil que B[2]: como mucho, la dificultad de resolver A es la de transformarlo en B a través de alguna $f \in F$, más la de resolver B, y esto no es más difícil que B (como "orden de magnitud" o "clase de complejidad") porque calcular $f(x)$ es "relativamente fácil".

[2]Es necesario hacer un pequeño esfuerzo para interiorizar la terminología clásica: aunque parezca poco intuitivo, es el problema "menor" que se reduce al "mayor", no al revés, como podría parecer más lógico.

Así que reducir un problema A a K ($A \leq_{rec} K$, donde se elige $\leq_{rec}$ como relación de reducción por el Teorema 9.1) demuestra que A es *como mucho RE*, es decir, seguramente es recursivamente enumerable (semidecidible) pero podría ser también recursivo)recursivo (decidible). En cambio, reducir K a A ($K \leq_{rec} A$) demuestra que A es *como poco RE*, es decir, puede ser recursivamente enumerable *pero seguramente no es recursivo (decidible)*. En este último caso A no sería recursivo porque, si lo fuera, entonces para resolver K para un input x bastaría transformar x en un input de A (con una función recursiva f) y luego resolver A para $f(x)$ (lo cual es, por hipótesis, recursivo). Esto implicaría que K también es recursivo, pero el Teorema 8.6 dice que no lo es.

Hecho 9.4 *Si un problema A se reduce a K ($A \leq_{rec} K$) entonces es como mucho recursivamente enumerable (podría ser recursivo).*

Hecho 9.5 *Si K se reduce a un problema A ($K \leq_{rec} A$) entonces A es como poco recursivamente enumerable, es decir, seguramente no es recursivo.*

Ejemplo 9.5 *Este ejemplo demuestra que el conjunto TOT de los índices de las funciones totales no es decidible (recursivo).*

$$TOT = \{\ x \mid \varphi_x \text{ es total}\ \}$$

Esto se hace reduciendo K a TOT (es decir, demostrando $K \leq_{rec} TOT$) a través de una función f que es calculable total, es decir, "fácil" con respecto a los niveles de dificultad que se están estudiando. Se define la siguiente función

$$\psi(x,y) = \begin{cases} 1 & \text{si } x \in K \\ \text{indefinida} & \text{en caso contrario} \end{cases}$$

que es calculable parcial porque K es semidecidible y ψ termina si y sólo si su primer argumento está en K. Por el Teorema del Parámetro, existe f calculable total y biunívoca tal que $\varphi_{f(x)}(y) = \psi(x,y)$. Es decir, para todo x existe una MT $M_{f(x)}$ que

- *termina para todo y devolviendo 1 (es decir, la función $\varphi_{f(x)}$ que calcula es la función constante $\lambda y.1$), si $x \in K$;*
- *no termina para ningún y (es decir, la función $\varphi_{f(x)}$ que calcula es la función constante $\lambda y.$indefinido), si $x \notin K$.*

Se da lo siguiente:

$$\begin{array}{lllll} x \in K & \Rightarrow & \varphi_{f(x)} = \lambda y.1 & \Rightarrow & \varphi_{f(x)} \text{ es total} \quad \Rightarrow \quad f(x) \in TOT \\ x \notin K & \Rightarrow & \varphi_{f(x)} = \lambda y.\text{indefinido} & \Rightarrow & \varphi_{f(x)} \text{ no es total} \quad \Rightarrow \quad f(x) \notin TOT \end{array}$$

es decir, x pertenece a K si y sólo si $f(x)$ pertenece a TOT, y f es calculable total (recursiva). Por definición, esto equivale a $K \leq_{rec} TOT$, por lo que TOT es por lo menos igual de difícil que K: no puede ser decidible.

9.3 Propiedades y Teorema de Rice

Entre los conjuntos de números naturales existen algunos conjuntos especiales, llamados *propiedades*, cuya característica es que incluyen exactamente los números derivados de la codificación de Máquinas de Turing que calculan ciertas funciones.

En ningún momento se ha dado un algoritmo específico para codificar Máquinas de Turing a números naturales, sino que simplemente se ha dicho que se trata de un procedimiento que puede llevarse a cabo *de manera efectiva*. Dada una MT M, es indiferente, en general, qué número n será su codificación: lo que interesa es saber que este n existe, y resulta únicamente de codificar M (y ninguna otra máquina).

Definición 9.3 (propiedades) *$I \subseteq \mathbb{N}$ es una* propiedad *si y sólo si*

$$\forall x \forall y.\ \ x \in I \wedge \varphi_x = \varphi_y \Rightarrow y \in I$$

es decir, si x e y son los índices de dos Máquinas de Turing distintas que calculan la misma función, entonces una propiedad contiene tanto x como y, o bien no contiene ninguno de los dos.

En otras palabras, una propiedad es una característica que una función calculable puede tener, y corresponde al conjunto de las funciones calculables (mejor dicho, de sus índices) que tienen esta característica.

Ejemplo 9.6 *Un ejemplo de propiedad es "ser una función total constante cuyo valor para todo input es un número par". El conjunto de índices que corresponde a esta propiedad es*

$$X = \{\ x \in \mathbb{N} \mid \forall z \in \mathbb{N}.\ \exists n \in \mathbb{P}.\ (M_x(z) = n)\ \}$$

X incluye los índices de todas las MTs que calculan el mismo valor para todo input, con la condición que este valor sea par.

Como siempre, todo lo dicho sobre las MTs puede traducirse a programas de algún lenguaje de programación, con su propia codificación a números naturales.

Ejemplo 9.7 *Si, en lugar de Máquinas de Turing, la codificación se refiriese a programas* WHILE*, la propiedad X del Ejemplo 9.6 incluiría rutinas como*

```
1 proc p2(n:int)
      returns int {
2   return 2;
3 }
```

```
1 proc p0a(n:int)
      returns int {
2   return 0;
3 }
```

```
1 proc p0b(n:int)
      returns int {
2   int i;
3   i := 100;
4   while (i>0) { i
        := i-1; }
5   return i;
6 }
```

Es preciso subrayar que, para cualquier propiedad Y (no sólo para X), las rutinas p0a *y* p0b *estarán ambas en Y o bien ambas fuera de Y, porque calculan la misma función calculable $\lambda x.0$.*

Ejemplo 9.8 *Sea C la siguiente clase de funciones calculables:*

$$C = \{\ f \mid \forall x \in \mathbb{N}.\ f(x) \text{ es par}\ \}$$

es decir, todas las funciones calculables totales cuyo output es par para todo input. La propiedad construida sobre C viene a ser

$$I_p = \{\ x \mid \exists f \in C.\ \varphi_x = f\ \}$$

e incluye todas las codificaciones de programas que calculan funciones de C. Es fácil ver que I_p es realmente una propiedad, al no contener nunca un índice sin contener también todos los infinitos índices de la misma función. Los programas que simplemente multiplican su input por un número par, como los siguientes, están en I_p.

```
1 proc q2(n:int)
      returns int {
2   return n*2
3 }
```

```
1 proc q4(n:int)
      returns int {
2   return n*4
3 }
```

```
1 proc q6(n:int)
      returns int {
2   return n*6
3 }
```

Pero también están en I_p muchos (infinitos) programas más complejos que siempre devuelven un número par, como la segunda rutina p *del Capítulo 1.*

Un conjunto $A \subseteq \mathbb{N}$ es una propiedad si identifica características *semánticas*, en ningún caso sintácticas, de los programas. Esto significa que A no puede contener sólo un índice entre x_1 y x_2 si los programas P_{x_1} y P_{x_2} no difieren por lo que calculan, sino únicamente por cómo la hacen.

Ejemplo 9.9 *Supongamos que, en este caso, la enumeración de las funciones calculables haya sido obtenida a partir de programas Java. De este modo, a todo $z \in \mathbb{N}$ corresponde, aparte de una función calculable φ_z, un programa Java P_z que la calcula. Sea A el conjunto de los x tales que*

- *φ_x es la función constante 0 ($\lambda y.0$); y*
- *el programa P_x tiene menos de 100 líneas de código.*

La característica de los elementos de A es entonces la conjunción de una propiedad semántica (que implemente la función constante 0) y una sintáctica (que tenga menos de 100 líneas de códigos).

Ahora, A no es una propiedad en el sentido de la Definición 9.3. Sean $\{i_0, i_1, ...\}$ los infinitos índices de $\lambda y.0$ (son infinitos por una fácil adaptación a Java del Hecho 8.1): sólo algunos de ellos corresponden a programas de menos de 100 líneas de código. Pongamos que el programa $P_{i_{12}}$ tenga 13 líneas de código, y que $P_{i_{26}}$ tenga 200. En este caso, no se cumple la definición de conjunto de índices para A al ser que $i_{12} \in A$, $\varphi_{i_{12}} = \varphi_{i_{26}}$ pero $i_{26} \notin A$.

Hecho 9.6 *Una propiedad representa características únicamente semánticas de programas (lo que hacen). Si también hay características sintácticas en la definición de cierto $A \subseteq \mathbb{N}$, lo normal es que no sea una propiedad (a no ser que la característica sintáctica sea trivial).*

Decimos "lo normal" porque también hay propiedades que tienen que ver con características sintácticas, pero esto sucede cuando éstas no son relevantes.

Ejemplo 9.10 *Supongamos que A sea el conjuntos de los z tales que*

- *φ_z es algún algoritmo complejo como la función de Ackermann; y*
- *el programa P_z que la calcula tiene menos de 10 caracteres de código.*

También en este caso, A se caracteriza por la conjunción de una característica semántica y una sintáctica, pero la segunda es trivial porque (presumiblemente) ningún programa que cumple la primera la satisface. Por lo tanto, A viene a ser el conjunto vacío, que sí es una propiedad.

¿Qué significa decidir la pertenencia de un índice a una propiedad? Significa que existe un algoritmo para decidir si un programa calcula o no cierta función, o una de las funciones de una cierta clase. Si las propiedades fueran conjuntos decidibles, podríamos contestar exactamente, en un tiempo finito, para todo programa y para todo input, preguntas como

- el programa ¿termina para todo input?
- el programa ¿termina para cierto input concreto?
- el programa ¿calcula la función f?
- etc.

porque cada una de estas preguntas está relacionada con una clase de funciones calculables (las que están definidas para todo input, las que están definidas para un input concreto, etc.).

Desafortunadamente, no es así: no sólo existen propiedades que no son decidibles, sino que *prácticamente todas* son indecidibles.

Teorema 9.2 *Sea I una propiedad tal que $\emptyset \neq I \neq \mathbb{N}$. Entonces $K \leq_{REC} I$ o bien $K \leq_{REC} \bar{I}$: K se reduce a ella o a su complementario.*

Demostración. Sea i_0 el índice de la función calculable φ_{i_0} que es siempre indefinida (sabemos que existe esta función). Si $i_0 \in \bar{I}$, entonces se demuestra que $K \leq_{REC} I$; si $i_0 \in I$, entonces un razonamiento simétrico lleva a demostrar que $K \leq_{REC} \bar{I}$. Sea $i_1 \in I$: este índice existe porque $I \neq \emptyset$ por hipótesis, y se da que $\varphi_{i_1} \neq \varphi_{i_0}$ porque $i_0 \in \bar{I}$ por hipótesis, y si φ_{i_1} fuera igual a φ_{i_0} entonces $i_1 \notin I$ por definición de propiedad. Usando el Teorema del Parámetro[3], se define una función

$$\varphi_{f(x)}(y) = \begin{cases} \varphi_{i_1}(y) & \text{si } x \in K \\ \text{indefinida} = \varphi_{i_0}(y) & \text{en caso contrario} \end{cases}$$

Ahora, con esta función se obtiene que

$$\begin{array}{lcll} x \in K & \Rightarrow & \varphi_{f(x)} = \varphi_{i_1} & \Rightarrow \quad f(x) \in I \\ x \notin K & \Rightarrow & \varphi_{f(x)} = \varphi_{i_0} & \Rightarrow \quad f(x) \notin I \end{array}$$

El último paso de cada una de las dos líneas se da porque la igualdad entre $\varphi_{f(x)}$ y φ_{i_1} implica que $f(x)$ está en I porque i_1 está, y la igualdad entre $\varphi_{f(x)}$

[3]Notamos que $\varphi_{f(x)}(y)$ es $\varphi_i(x, y)$, y que f es $\lambda x.s_1^1(i, x)$.

y φ_{i_0} implica que $f(x)$ no está en I porque i_0 no está. Con este procedimiento se ha obtenido una función calculable total que transforma la decisión de K en la decisión de I, así que se demuestra, por reducción, que cualquier propiedad I no trivial es indecidible.

Si en cambio se llega a demostrar que $K \leq_{REC} \bar{I}$ entonces I sigue siendo indecidible porque, si fuera decidible, $\bar{I}$ también lo sería (Ejercicio 8.12), por lo que K no podría reducirse a él. □

El siguiente corolario pone unos límites muy duros a lo que se puede demostrar acerca de las funciones calculables, y es probablemente el Teorema más importante de la Teoría de la Computabilidad.

Teorema 9.3 (Rice) *Sea C una clase de funciones calculables parciales. El conjunto $\{ x \mid \exists f \in C.\ \varphi_x = f \}$ es decidible si y sólo es vacío o es todo $\mathbb{N}$.*

Demostración. Fácil dado el Teorema 9.2. □

Entre las clases de funciones calculables que no son decidibles se encuentran

- $K_1 = \{ x \mid$ el dominio de φ_x no es $\emptyset \}$
- $FIN = \{ x \mid$ el dominio de φ_x es finito $\}$
- $INF = \{ x \mid$ el dominio de φ_x es infinito $\}$
- $TOT = \{ x \mid \varphi_x$ es total $\}$
- $CONST = \{ x \mid \varphi_x$ es total y constante $\}$
- $RECUR = \{ x \mid$ el dominio de φ_x es recursivo $\}$

Como última observación, estos conjuntos, con la excepción de K_1 que sí lo es, ni siquiera son recursivamente enumerables, así que no se puede decidirlos ni semidecidirlos.

9.4 Resumen

Resumen 9.1 *Dada una clase de complejidad $\mathcal{D}$, los problemas $\mathcal{D}$-completos son "los más difíciles" dentro de esta clase.*

Resumen 9.2 *El conjunto K, visto como problema, es RE-completo.*

Resumen 9.3 *Si un problema A se reduce a K entonces es como mucho recursivamente enumerable, y podría ser recursivo. Si K se reduce a A, entonces A es como poco recursivamente enumerable, es decir, seguramente no es recursivo.*

Resumen 9.4 *Una propiedad es un conjunto de índices que representa características únicamente semánticas de programas o funciones calculables.*

Resumen 9.5 *Las propiedades no triviales no son decidibles.*

Resumen 9.6 *No se puede decidir si un programa o función calculable pertenece a cierta clase no trivial de funciones.*

9.5 Ejercicios

Ejercicio 9.1 *Demostrar el Hecho 9.1.*

Ejercicio 9.2 *Demostrar el Hecho 9.2.*

Ejercicio 9.3 *Demostrar que no es recursivo el predicado*

$$P(x) = \begin{cases} 1 & \textit{si el codominio de } \varphi_x \textit{ es infinito} \\ 0 & \textit{en caso contrario} \end{cases}$$

Ejercicio 9.4 *Demostrar que no es recursivo el predicado*

$$P(x) = \begin{cases} 1 & \textit{si } \varphi_x \textit{ es una función total} \\ 0 & \textit{en caso contrario} \end{cases}$$

Ejercicio 9.5 *Demostrar que no es recursivo el predicado*

$$P(x,y,z) = \begin{cases} 1 & \textit{si } \varphi_x(y) = z \\ 0 & \textit{en caso contrario} \end{cases}$$

Ejercicio 9.6 *Demostrar que no es recursivo el predicado*

$$P(x) = \begin{cases} 1 & \textit{si } \varphi_x \textit{ es inyectiva} \\ 0 & \textit{en caso contrario} \end{cases}$$

Ejercicio 9.7 *Sea A recursivamente enumerable. Demostrar que*

$$B = \bigvee_{n \in A} W_n$$

es recursivamente enumerable, donde W_n es el dominio de φ_n, es decir, los valores de input para los que φ_n está definida o converge.

Ejercicio 9.8 *Estudiar si es recursivamente enumerable el siguiente conjunto o su complementario:*

$$A = \{\ x \mid \varphi_x \text{ es inyectiva } \}$$

Ejercicio 9.9 *Estudiar si es recursivamente enumerable el siguiente conjunto o su complementario:*

$$TOT = \{\ x \mid \varphi_x \text{ es total } \}$$

Ejercicio 9.10 *Estudiar si es recursivamente enumerable el siguiente conjunto o su complementario:*

$$A = \{\ x \mid \text{el dominio de } \varphi_x \text{ es infinito } \}$$

Ejercicio 9.11 *Demostrar que el conjunto A del Ejemplo 9.6 es un conjunto de índices.*

Ejercicio 9.12 *Demostrar el Teorema de Rice (Teorema 9.3).*

Ejercicio 9.13 *Demostrar que K_1 es recursivamente enumerable.*

Ejercicio 9.14 *Decir por qué a partir de la solución del ejercicio 9.6 se demuestra directamente que*

$$A = \{\ x \mid \varphi_x \text{ es inyectiva } \}$$

no es recursivamente enumerable.

Ejercicio 9.15 *Se ha dicho que el conjunto RECUR no es ni recursivo ni recursivamente enumerable. ¿Qué se puede decir del siguiente conjunto?*

$$RECURENUM = \{\ x \mid \text{el dominio de } \varphi_x \text{ es recursivamente enumerable } \}$$

Parte III

Consecuencias teóricas y prácticas

Capítulo 10

Consecuencias del Teorema de Rice

La consecuencia más directa del Teorema de Rice es que, en general, no existe una forma de decidir algorítmicamente si un programa calcula o no cierta función f, es decir, si tiene o no cierto comportamiento. Esto sucede porque una propiedad I identifica todos los índices que corresponden a una cierta función calculable, o, más en general, a una cierta clase de funciones calculables, así que la indecidibilidad de toda propiedad que no sea trivial (es decir, que no sea $\emptyset$ ni $\mathbb{N}$) implica que no se puede decidir si un programa[1] calcula dicha función.

Hecho 10.1 *Al ser indecidible toda propiedad que no sea trivial, no se puede decidir, en el caso general, si un programa calcula o no cierta función.*

Decir que un programa calcula una función f significa afirmar que el output es $f(x)$ para todo input $x \in dom(f)$, incluyendo estados de error (que se pueden considerar como un caso especial de output) u otros comportamientos excepcionales. En realidad, decir que no se puede decidir si un programa tiene cierto comportamiento no da del todo la idea de nuestra impotencia: en muchísimos casos, es imposible incluso decidir si un programa tiene o no cierta *característica* semántica.

El teorema de Rice dice que las características semánticas no triviales son indecidibles. En cambio, hay características *sintácticas* que son decidibles: por ejemplo, la longitud de un programa (el número de líneas o de caracteres del

[1] Como siempre, se "confunde" programas con MTs sin que surja ningún problema.

código fuente) o el número de variables que usa, que se pueden calcular muy fácilmente inspeccionando el código. Estas características no son semánticas porque no se refieren a *lo que hace* un programa, sino a *cómo es*. Por esto, en general, no hay correspondencia con las propiedades de la Definición 9.3.

El resultado de indecidibilidad de todas las propiedades no triviales tiene una implicación: si se quiere investigar lo que hace un programa P (por ejemplo, para ver si calcula o no cierta función, o si tiene algún fallo), no existe ninguna herramienta de análisis que *decida* (es decir, que diga "sí" cuando tiene que decir "sí", y "no" cuando tiene que decir "no", y termine siempre) en todos los casos si P tiene el comportamiento esperado. Por lo tanto, cualquier tipo de análisis que se pueda plantear tendrá que renunciar a dar una respuesta correcta en la totalidad de los casos.

Sin embargo, analizar el comportamiento de un programa es una tarea cada vez más importante en la Ingeniería del Software, porque la sociedad en la que vivimos se apoya cada vez más en sistemas informático complejos, y las consecuencias de un fallo en estos sistemas pueden ser desastrosas. Se entiende, por lo tanto, la necesidad de producir metodologías de análisis de programas y sistemas informáticos que sepan minimizar las consecuencias negativas del Teorema de Rice y proporcionen resultados útiles a nivel práctico. Dos de los enfoques principales en este sentido son:

- El *Análisis Dinámico*: se ejecuta P para una serie de valores de input (*test cases* o *casos de prueba*) para ver cómo se comporta. Este enfoque se llama también *testing* y su principal limitación es que no cubre todos los casos al ser imposible probar un programa para todos los valores de input posibles. Experimentamos esta limitación a diario, ya que todo programa no trivial tiene fallos (*bugs*), y si esto pasa es porque los fallos no han sido detectados durante la fase de *testing*. Además, la presencia de bugs no es meramente una cuestión que se pueda desestimar porque la probabilidad de dar con uno es muy baja: hay situaciones en las que algún malintencionado busca justamente ese 0.01% de probabilidad de que ocurra algo no previsto por el programador, con el objetivo de vulnerar la seguridad de un sistema complejo.

- El *Análisis Estático*: se estudia la estructura de P (su código) sin ejecutarlo, intentando inferir información útil sobre su funcionamiento. Este segundo enfoque va a ser el tema del resto de este capítulo.

Hecho 10.2 *El análisis dinámico no puede cubrir todos los casos (valores de input) de uso (ejecución) de un programa.*

10.1 Analizadores estáticos

Un *analizador estático* es un programa A que trabaja con otro programa P (que puede estar escrito en el mismo lenguaje o en un lenguaje distinto) proporcionando información sobre él, y lo hace *sin ejecutarlo*, sino simplemente estudiando su código[2]. ¿Por qué no ejecuta los programas? Está claro que ejecutar un programa, como en el análisis dinámico, diría con toda la precisión posible lo que este programa hace, pero sólo lo diría para un input concreto. Para obtener información sobre todos los casos posibles se debería considerar un número infinito (o, en todo caso, demasiado grande) de valores de input.

Ambos enfoques de análisis, tanto el estático como el dinámico, sufren las consecuencias del Teorema de Rice:

- El dinámico, porque no puede cubrir todos los casos;
- El estático, porque no se puede hacer que, en el caso general, el analizador diga siempre y correctamente (decida) si un programa tiene o no cierta característica semántica. Por lo tanto, un procedimiento efectivo de análisis estático *tiene que equivocarse* en algunos casos.

Hecho 10.3 *El análisis estático cubre todos los casos de uso de un programa, pero el resultado del análisis es necesariamente aproximado: el analizador se equivoca siempre alguna vez (para algún programa analizado).*

A nivel industrial, lo que se usa es principalmente el análisis dinámico, porque es conceptualmente más fácil y encuentra la mayoría de los errores. En cambio, el análisis estático se empieza a considerar más como herramienta para la verificación de código: se usa en muchos compiladores y hay mucha investigación sobre la implementación de técnicas de análisis estático de propiedades complejas. Su principal debilidad, aparte el Teorema de Rice, es que es muy difícil definir e implementar buenos analizadores para propiedades complejas que aseguren la corrección de los programas analizados y sean razonablemente precisos y eficientes. Por otro lado, su gran ventaja, como veremos, es que, si está bien implementado, *sólo se equivoca en un sentido* y, si llega a decir que un programa tiene una propiedad, es que realmente la tiene.

Hecho 10.4 *Un analizador estático de alguna propiedad tiene que equivocarse alguna vez sobre la pregunta "el programa P, ¿tiene la propiedad π?". Pero es*

[2]Normalmente A estudia el *código fuente* de P, pero no habría ninguna diferencia conceptual importante si estudiase su código máquina o algún tipo de código intermedio.

posible implementarlo de manera que siempre se euivoque en el mismo sentido: es decir, que nunca diga "sí" cuando la respuesta correcta para P es "no", aunque puede decir "no" aunque la respuesta correcta sea "sí".

10.2 Ejemplos de analizadores estáticos

Este capítulo se centra en el análisis estático y en analizadores que, dado un programa P, intentan contestar "sí" o "no" dependiendo de si P tiene o no cierta propiedad. Algunos ejemplos de analizadores estáticos son:

Ejemplo 10.1 (Type checker) *En un lenguaje de programación, los tipos sirven para garantizar que nunca se van a dan ciertos errores a tiempo de ejecución. Un* type checker *se ocupa del* chequeo de tipificación*: averigua que no hay incoherencias entre los datos que el programa maneja.*

En algunos lenguajes funcionales, como Haskell, el sistema de tipos *es tan complejo y expresivo que puede resultar imposible, en un tiempo finito, descartar por errores de tipo exclusivamente los programas que realmente pueden dar errores a tiempo de ejecución, es decir, que el type checker también descartará programas que nunca dan error. Para evitar esto, a veces se toma el camino peligroso de permitir (a través de la activación de ciertos mecanismos a tiempo de compilación) que el control de tipos pueda* no terminar*. Esto equivale a decir dos cosas: (1) que ciertos sistemas de tipos definen propiedades indecidibles; y (2) que, en general, la propiedad "nunca da errores a tiempo de ejecución" es indecidible.*

Ejemplo 10.2 (Analizador de terminación) *Como ha quedado claro a lo largo de este libro, la terminación de un programa para todo input no se puede decidir en el caso general. Entonces un* analizador de terminación *tendrá que* aproximar *el resultado, es decir, equivocarse en algunos casos.*

Más concretamente, un analizador de terminación intentará contestar "sí" para todos los programas que terminan para todo input, y "no" para todos los programas que no terminan para algún input. Al ser la terminación una propiedad indecidible, el analizador tendrá que contestar "no" incluso para algunos programas que en realidad terminan siempre.

10.3 Aproximar propiedades indecidibles

Aún siendo normalmente indecidibles, las propiedades tienen un interés muy grande en el ámbito de la Ingeniería del Software. Para usar un ejemplo ya

bien conocido, saber que un programa termina para todo posible input permite ejecutarlo sin temor a que "se quede colgado", lo que puede ser una situación extremadamente indeseable en ciertos contextos.

Hay dos ingredientes:

- propiedades interesantes que nos gustaría decir sobre los programas; y
- la imposibilidad matemática de hacerlo.

Lo único que queda es aceptar que el analizador se equivoque algunas veces e intentar sacar en todo caso el máximo provecho de su trabajo.

Supongamos que un analizador A para una propiedad π se pueda equivocar en los dos sentidos: (a) a veces contesta "sí" para un programa P que no tiene la propiedad π, y (b) a veces contesta "no" para un programa que sí la tiene. Si, como normalmente sucede, π es una característica deseada de un programa (por ejemplo, que termine siempre, o que no genere errores a tiempo de ejecución), lo que se intenta es producir programas que tienen esta propiedad. Por lo tanto, la situación (b) es una *falsa alarma*: se rechaza un programa que en realidad no es problemático. En cambio, el caso (a) implica una *exceso de confianza* al aceptar un programa que puede dar problemas. Si tanto (a) como (b) pudiese pasar, no podríamos fiarnos de la respuesta de A, y su trabajo sería bastante inútil si lo que se busca es algún tipo de garantía. En cambio, si se consiguiera que A sólo se equivocase en un sentido, entonces su respuesta sería en todo caso una respuesta útil: saber que A sólo se equivoca en el sentido (b), dando falsas alarmas de vez en cuando, entonces una respuesta "sí" de su parte permite confiar en que P *tiene* la propiedad π.

Pensemos en la propiedad de *terminación para cualquier input*: se trata de estudiar el conjunto

$$\{\ x \mid \forall y.\ \varphi_x(y) \text{ termina }\}$$

Lo ideal sería que A dijera que "sí" para todo programa "bueno" que termina siempre, y que "no" para todo programa "malo" que a veces no termina, es decir, que *decida* la pertenencia al conjunto. Al ser esto imposible, hay que estar dispuestos a *juzgar erróneamente* algunos programas, es decir, *equivocarse*:

- considerar como malos algunos programas que siempre terminan (falsa alarma), y/o
- considerar como buenos algunos programas que a veces no terminan (exceso de confianza).

En principio el analizador podría equivocarse en los dos sentidos, pero no suele ser una buena opción permitir que esto suceda:

- si el desarrollador ve rechazado como "malo" su programa, aunque en realidad siempre termina, éste tendrá un trabajo extra por hacer (modificar el programa hasta que A lo acepte), pero esto no supone mayor problema con respecto a la calidad del producto final;
- en cambio, si A considera como bueno un programa que puede no terminar, este puede ser distribuido con consiguiente posibilidad de comportamientos indeseados, que pueden costar trabajo, fallos de seguridad o dinero.

Hay una diferencia de importancia entre los dos problemas: una falsa alarma es un mal relativamente menor en cuanto supone la necesidad de mejorar el código, mientras que un exceso de confianza puede suponer un importante fallo de seguridad. Este ejemplo gráfico nos ayuda a entender este concepto:

Ejemplo 10.3 *En un universo paralelo, unos agricultores japoneses han conseguido la manzana* superfuji*, la más sabrosa que nunca haya existido, que tiene un sabor único, y también un precio deshorbitado debido a la gran dificultad y coste de producción. Cada gramo de esta manzana* superfuji *tiene un enorme valor por el placer que puede dar a los paladares más selectos, así que hay que apovechar cada mínima porción del fruto.*

Sin embargo, la super-manzana superfuji *tiene un "pero": comer una parte incluso pequeña de su piel es mortal para cualquier hombre. Por lo tanto, se necesitan cuchillos de grandísima precisión que*

- *desperdicien menos posible la parte buena de la* superfuji *(la pulpa);*
- *nunca dejen algo de piel en lo que se va a comer; y*
- *permitan pelar la manzana en un tiempo razonable.*

La división entre pulpa y piel es la propiedad de interés: los programas que la cumplen son la pulpa, los otros son la piel. El cuchillo es el analizador: en algunos casos puede descartar programas buenos (es inevitable que alguna molécula de la pulpa se pierda), pero nunca aceptará programas malos, y es un buen analizador en la medida en que es poca la pulpa que desperdicia. Por último, el analizador tiene que actuar no sólo en un tiempo finito, sino también con cierta eficiencia, para que se pueda aplicar a programas de tamaño respetable.

10.4 Otros ejemplos de análisis estático

Los ejemplos de esta sección se refieren al lenguaje de programación Java. Se trata de características que, aparentemente, tienen que ver con la estructura sintáctica de un programa, pero basta aplicar unas transformaciones sencillas para reducirlas a características totalmente semánticas. Una vez más, las propiedades que se discuten aquí son indecidibles.

10.4.1 División por cero

Un típico error que se puede dar ejecutando un programa es que una división entre números tenga denominador 0. Normalmente, una división por cero implica la terminación abrupta del programa. Si, con una sencilla transformación del código, se asocia a la división por cero un valor especial de output v_0, la propiedad estudiada "el programa nunca ejecuta una división por cero" se reduce a "el programa nunca devuelve el valor v_0", que es claramente una característica semántica porque corresponde a una propiedad. Concretamente, esta transformación de código se puede obtener insertando cada división en un test que comprueba si el denominador es cero y, en caso positivo, hace que el programa termine devolviendo v_0. En el caso de java, este valor especial se puede modelizar eficazmente con una excepción.

Ejemplo 10.4 *El código*

```
1  x = y/z;
```

se puede transformar en

```
1  if (z==0) {
2    throw new DivisionByZero();
3  } else {
4    x = y/z;
5  }
```

obviamente asegurando que la excepción implique la terminación del programa.

Se trata de otra propiedad indecidible, al ser imposible, en el caso general, saber si durante la ejecución un denominador tendrá este valor. Es interesante notar que, en este caso, el compilador javac[3] decide no analizar el programa con respecto a este error, dejando al programador la tarea de evitar que esto ocurra.

[3]Versión: javac 1.6.0_30 en Ubuntu.

Sin embargo, se podría implementar un analizador estático que descartara cualquier programa que puede en algunos casos ejecutar una división por cero. La indecibilidad de esta propiedad implica que dicho analizador descartará programas que nunca ejecutan una división por cero.

Ejemplo 10.5 *El siguiente programa puede ejecutar una división por cero, al tener la variable* `j` *un valor desconocido (porque es un input) en la línea 1:*

```
1  k = 0;
2  for (int i=1; i<=10; i++) k += 100/(j+i);
```

En cambio, en este código el denominador es positivo para todo posible input:

```
1  k = 0;
2  for (int i=1; i<=10; i++) k += 100/(j*j+i);
```

El analizador tendría que ser lo bastante "inteligente" como para darse cuenta de la positividad del denominador y, por lo tanto, aceptar el segundo programa. En todo caso se pide al analizador que rechace el primer programa como potencialmente peligroso.

10.4.2 Acceso a campos de punteros "null"

Una propiedad de alguna forma parecida a la posibilidad de ejecutar divisiones por cero es el acceso a campo de punteros **null**: en ambos casos, hay un valor de una variable que genera un error. Esta propiedad también es indecidible, y javac tampoco la estudia: para averiguarlo basta escribir un ejemplo muy sencillo de programa que compila correctamente pero da error a tiempo de ejecución porque se ejecuta algo como

```
1 x = null;
2 y = x.f;
```

En este caso, la máquina virtual de Java lanza una excepción NullPointerException , que se puede considerar como el valor de output que un programa nunca debería devolver.

Existen muy buenas razones para implementar un análisis de "nulidad" que intenta garantizar la siguiente característica semántica: nunca se accede en lectura al valor `x.f` si la variable `x` es **null**. Además de evitar el lanzamiento de NullPointerException, que ya de por sí es una razón suficiente para analizar esta propiedad (es una de las causas más comunes de terminación abrupta en las aplicaciones de nuestros dispositivos móviles), saber que una variable nunca es **null** en cierto punto del código ayuda ciertos procesos de optimización a tiempo

de compilación, y es una información muy útil para mejorar los resultados de otros análisis.

Ejemplo 10.6 *Es bastante común que un programador ponga un test* **if** (x!= **null**) *antes de ejecutar algún comando, aunque su intención es que la variable siempre apunte a algún objeto en el heap. Si es posible garantizar que la condición siempre se cumple, el compilador podría eliminar el test y la rama "else", produciendo así un programa ejecutable más sencillo, de menor tamaño y más eficiente.*

10.4.3 Acceso a variables no inicializadas

Consideremos el siguiente método main escrito en Java:

```
1 public class varInit {
2   public static void main(String[] str) {
3     int i=str.length;
4     int j, k;
5     if (i>10) k=((i>5)?i:j)+1;
6     else k=i;
7   }
8 }
```

Al compilarlo en javac se obtiene el siguiente error:

```
varInit.java:5: variable j might not have been initialized
      if (i>10) k=((i>5)?i:j)+1;
                           ^
1 error
```

que significa que podría haber un uso de la variable j antes de su inicialización. Sin embargo, al mirar más detenidamente el código nos damos cuenta de que esto nunca va a pasar, ya que la rama "then" del if-then-else externo sólo se ejecuta si i>10, y en este caso también se cumple la condición i>5, lo que asegura que a k se le asigna el valor i+1 en lugar de j+1.

La característica semántica de programas que se estudia aquí es "en ninguna ejecución usa una variable antes de que ésta haya sido inicializada". Al ser también esta propiedad indecidible, el compilador tiene que optar por una aproximación: básicamente, rechazará todo programa en el que, mirando el flujo de control, sea posible llegar a un comando que usa una variable sin haber pasado por ningún comando que la inicializa. En este programa, coger la rama "then" del if-then-else exterior y la rama "else" de la sentencia condicional

interior lleva a esta posibilidad, y es lo que hace que el compilador lo rechace. Sin embargo, un análisis más detallado de las condiciones sobre `i` llevaría a detectar que esta combinación nunca se da, pero esto requiere un análisis demasiado complejo y de difícil implementación para que se incorpore a una herramienta ya muy compleja como un compilador. De hecho, existen técnicas para desminuir el número de programas que son rechazados a pesar de no ser realmente problemáticos, pero la Teoría de la Computabilidad dice que la solución definitiva no existe. El compilador `javac` decide implementar un análisis bastante básico sacrificando la precisión a la eficiencia y la sencillez.

10.4.4 Eliminación de Código Muerto (Dead Code)

El ejemplo 10.6 ilustra un caso en que una parte del código nunca se ejecuta: la rama "else" de la sentencia condicional. Esta rama es "código muerto", y se puede eliminar sin afectar el programa, resultando además en un programa ejecutable más pequeño y eficiente. En general, hay numerosas técnicas de análisis implementadas como partes de los compiladores que tienen como objetivo demostrar que ciertas porciones de código son muertas y se pueden eliminar. Esto conlleva beneficios como por ejemplo un menor tamaño del código ejecutable. Por ejemplo, en el siguiente código Java

```
1 double d = sqrt(2);
2 if (d > 5) {
3   a = 2;
4 }
```

la línea 3 nunca llega a ejecutarse, por lo que la sentencia condicional se podría eliminar. Sin embargo, puede ser difícil detectar esta situación porque se necesitaría dotar el analizador de operaciónes sobre números no enteros, cosa que puede llegar a ser muy costosa.

10.5 Resumen

Resumen 10.1 *Dado que toda propiedad no trivial es indecidible, no se puede decidir, en el caso general, si un programa calcula o no cierta función.*

Resumen 10.2 *Dado que es imposible en general analizar completa y exactamente un programa, y sin embargo el análisis de programas tiene una importancia muy grande en la Ingeniería del Software, se intenta definir metodologías*

de análisis que por lo menos sean útiles en la práctica: entre ellas, el análisis dinámico y el análisis estático.

Resumen 10.3 *El análisis dinámico no puede cubrir todos los casos de ejecución de un programa.*

Resumen 10.4 *Entre las propiedades sencillas pero interesantes de programas que se quieren estudiar, están la posibilidad de dividir por* 0*, el acceso a campos de punteros null, el acceso a variables no inicializadas, la presencia de código muerto, etc. Todas estas propiedades son indecidibles, pero existen herramientas de análisis estático que las aproximan.*

Resumen 10.5 *Aproximar una propiedad indecidible significa permitir que el analizador se equivoque en un número limitado de casos, pero garantizar que sólo se equivocará en un sentido. En general, se aceptan falsas alarmas pero se quieren evitar a toda cuesta excesos de confianza.*

10.6 Ejercicios

Ejercicio 10.1 *Dado el código*

```
1  int j = 0, k = 0;
2  if (i%2 != 0) i++;
3  while (i<100) {
4    k = k+j/(i-1);
5    j++;
6    i += 2;
7  }
```

donde i *está inicializada a un valor desconocido, decir si es posible que se ejecute una división con denominador* 0*.*

Ejercicio 10.2 *Dado el código*

```
1  int j = 0;
2  int k = i/2;        // div. entera
3  if (i>100) {
4    j = 100/(k+1);
5  } else {
6    j = 100/k;
7  }
```

donde i *está inicializada a un valor estrictamente positivo, decir si es posible que se ejecute una división con denominador* 0.

Ejercicio 10.3 *Dado el código*

```
1   int n;
2   while (i!=0) {
3     if (i>0) {
4       n = 100/i;
5       i--;
6     } else {
7       n = -100/i;
8       i++
9     }
10  }
```

donde i *está inicializada a un valor desconocido, decir si es posible que se ejecute una división con denominador* 0.

Ejercicio 10.4 *Dado el código*

```
1   Clase1 x = new Clase1();
2   while (x != null) {
3     x.f = new Clase1();
4     x.g = x.f;
5     x = x.f;
6   }
```

decir si es posible que se lance una NullPointerException *durante la ejecución. Las constructoras no hacen nada.*

Ejercicio 10.5 *Dado el código*

```
1   Clase2 x = new Clase2();
2   Clase2 y = new Clase2();
3   Clase2 z;
4   x.f = y;
5   if (i%2==0) {
6   z = x;
7   } else z = y;
8     int k = z.f.data;
```

decir si es posible que se lance una NullPointerException *durante la ejecución. La clase* Clase2 *tiene campos* f *de tipo* Clase2 *y* data *de tipo* **int**. *Las constructoras no hacen nada, e* i *tiene algún valor entero desconocido.*

Ejercicio 10.6 *Dado el código*

```
1  int j;
2  for (int i=0; i<10; i++) {
3    if (i>0) {
4      j = 100;
5    } else j++;
6  }
```

decir si es posible que se acceda a la variable `j` *sin que esté inicializada.*

Ejercicio 10.7 *Dado el código*

```
1  int j,k;
2  if (i%2 == 0) {
3    if (2*i==10) { k = j; }
4    else k = 0;
5  }
```

donde `i` *se supone que sí está inicializada a algún valor desconocido, decir si es posible que se acceda a la variable* `j` *sin que esté inicializada.*

Ejercicio 10.8 *Dado el código*

```
1  int j = 10;
2  int k;
3  while (j!=0 && k!=0) {
4    j--;
5    i = i*2;
6    if (i>100000) {
7      k = j+1;
8    }
9  }
```

donde `i` *se supone que sí está inicializada a algún valor desconocido no negativo, decir si es posible que se acceda a la variable* `k` *sin que esté inicializada.*

Ejercicio 10.9 *Dado el programa*

```
1  i = i*i;
2  int j = 10;
3  int x;
4  int y = 0;
5  if (j>i) { x = j; } else { x = i; }
6  while (x-i>10) {
7    y++;
```

```
8   }
```

donde i *tiene inicialmente algún valor entero desconocido, decir si la línea 6 es o no código muerto y justificar la respuesta.*

Ejercicio 10.10 *Dado el programa*

```
1   int j=1;
2   while (i>10) {
3     if (j!=0) {
4       j = j-2;
5       i = i+2;
6     } else {
7       j++;
8       i--;
9     }
10  }
```

donde i *tiene inicialmente algún valor entero desconocido, decir si las líneas 7 y 8 son o no código muerto y justificar la respuesta.*

Capítulo 11

Construyendo analizadores

Este capítulo no quiere ser ni mucho menos una tratación completa de este tema. Entender esta temática en profundidad requiere un estudio específico que se sale, y mucho, de los objetivos de este libro. Para el lector interesado, un buen punto de partida es el trabajo de Flemming Nielson, Hanne R. Nielson y Chris Hankin [14].

El recorrido hecho en los capítulos anteriores nos ha llevado a ver que existen propiedades indecidibles, y que tienen que ver con la vida real del informático. De hecho, estas propiedades son prácticamente todas las características semánticas que se pueden encontrar interesantes a la hora de estudiar un programa y lo que calcula. Se ha visto que cualquier analizador (estático) que intente decidir estas propiedades tendrá que equivocarse algunas veces, pero, en principio, es posible hacerlo de manera que siempre se equivoque "en el mismo sentido" (normalmente, sólo *falsas alarmas* y ningún *exceso de confianza*).

Queda por ver cómo se puede plantear, en general y de forma muy introductoria, el diseño de un analizador. Este capítulo plantea algunas ideas, por lo menos en lo que se refiere a algunas formas específicas de construir analizadores[1].

[1]Se va a dar un enfoque que está relacionado con la teoría de la Interpretación Abstracta [5]; está claro que no es el único enfoque posible, pero no sería realista intentar cubrir todos los aspectos de esta temática, así que esta discusión se limita a uno de ellos.

11.1 La semántica de un lenguaje sencillo

El lector que esté familiarizado con el concepto de *semántica* de una lenguaje de programación, y, más concretamente, de *semántica denotacional*, no necesita leer esta sección muy detenidamente porque su contenido no le va a sorprender en absoluto. Para los demás, el principal objetivo de introducir la semántica denotacional de un lenguaje imperativo sencillo es sentar las bases para los conceptos explicados en la Sección 11.2.

Una semántica sirve básicamente para dar un significado a los programas escritos en cierto lenguaje de programación. Es lo que tiene que conocer un compilador a la hora de producir código ejecutable; es más, el compilador tiene que conocer y aplicar correctamente tanto la semántica del lenguaje original como la del lenguaje de máquina. Y es también lo que (muy a menudo informalmente) aplica un informático a la hora de producir código o estudiar lo que hace un programa.

Esta sección introduce la semántica del lenguaje WHILE, limitándose al nivel de formalización estrictamente necesario para entender los conceptos básicos. WHILE es un lenguaje imperativo sencillo que únicamente usa datos numéricos y booleanos. La sintaxis del lenguaje se encuentra en la Sección 5.2, pero allí no se dio la definición formal de lo que significa ejecutar un programa, limitándo la discusión a decir que el significado se correspondía con la intuición y cualquier informático sería capaz de entenderlo.

Dado un programa WHILE, ¿cómo se puede describir formalmente lo que sucede en cada paso de su ejecución? Definiendo un estado de ejecución y describiendo cómo afecta al estado todo tipo de sentencia que se pueda escribir. El *estado* es la información necesaria para saber cómo actúa un programa, y se va modificando durante la ejecución a partir de un estado inicial. La *semántica* de una sentencia describe el efecto de una sentencia sobre el estado. Por ejemplo, la sentencia `x:=4` cambia el valor de `x` en el estado, poniéndolo a 4. La información que tiene que contener el estado es el valor actual de cada variable.

Definición 11.1 *Un* estado E *de un programa* WHILE P *es una función que para todo nombre de variable definida en el programa devuelve un valor: para toda variable* v, $E(v) \in \mathbb{Z} \cup \{true, false\}$ *(un número entero o un valor de verdad, dependiendo del tipo de* v*).*

En el lenguaje WHILE, un programa es una serie de declaraciones de rutinas, luego unas declaraciones de variables y finalmente un bloque de código (una sentencia) principal que es lo que se va a ejecutar, y que, aproximadamente,

juega el mismo papel que el método **main** en Java. Un ejemplo de esto es el programa sencillo del Capítulo 1, donde la sentencia principal empieza en la línea 7 con el **begin** después de declarar dos variables en la línea 6. En este capítulo no se considera la llamada de rutinas porque complicaría bastante las cosas: en cambio, se asume que todo el programa está contenido en el bloque principal (las declaraciones de variables y el código entre **begin** y **end**).

Ejemplo 11.1 *Se onsidere el siguiente programa P:*

```
1 var x:int, y:int;
2 begin
3   x := 1;
4   y := 2;
5   while (x<=10) do { x := x+1 }
6 end
```

Este programa sólo incluye la declaración de dos variables y el bloque principal. Después de ejecutar las primeras dos sentencias, y de comprobar si la condición del bucle es verdad, el estado E es tal que (1) $E(\mathsf{x}) = 1$; y (2) $E(\mathsf{y}) = 2$. Lo siguiente será ejecutar el cuerpo del bucle. El estado final E' será tal que $E'(\mathsf{x}) = 11$ y $E'(\mathsf{y}) = 2$.

El *estado inicial* es el estado que se produce a partir de las declaraciones de variables antes de la sentencia principal. Para no complicar demasiado las cosas, se supone que

- si dentro del código se declaran otras variables, el estado se ensancha automáticamente para incluir sus valores[2];
- al declarar una variable, su valor inicial es 0 si es de tipo **int**, *false* si es de tipo **bool**;
- no se declaran en sitios distintos variables con el mismo nombre.

El siguiente paso es definir si y cómo cada sentencia modifica el estado en el que es ejecutada. Se empieza definiendo cómo se evalúan las expresiones, tanto aritméticas como booleanas: dada una expresión *exp*, $[\![exp]\!](E)$ es el valor de *exp* calculado en el estado E, es decir, el *significado* de la expresión en E. Así que el valor de una constante numérica no es otra cosa que el número correspondiente:

$$[\![n]\!](E) = n \text{ (el "verdadero" número } n)$$

[2]Obtener esto no es del todo trivial, pero lo damos por descontado al no estar interesados en los detalles.

El valor de una variable es el que está almacenado en el estado:

$$[\![x]\!](E) = E(x)$$

Las expresiones numéricas obtenidas combinando expresiones más pequeñas se evalúan calculando recursivamente el valor de las sub-expresiones y luego aplicando la correspondiente operación aritmética:

$$\begin{array}{rcl} [\![-exp]\!](E) & = & -([\![exp]\!](E)) \\ [\![exp_1 + exp_2]\!](E) & = & ([\![exp_1]\!](E)) + ([\![exp_2]\!](E)) \\ [\![exp_1 - exp_2]\!](E) & = & ([\![exp_1]\!](E)) - ([\![exp_2]\!](E)) \\ [\![exp_1 * exp_2]\!](E) & = & ([\![exp_1]\!](E)) * ([\![exp_2]\!](E)) \\ [\![exp_1 \,/\, exp_2]\!](E) & = & ([\![exp_1]\!](E)) \,/\, ([\![exp_2]\!](E)) \end{array}$$

Los valores booleanos se pueden representar con números, como en el lenguaje C: 0 equivale a "falso" y todo número que no sea 0 equivale a "verdadero". Los operadores booleanos se aplican de manera consecuente: por ejemplo, la conjunción es el producto porque $0 * n = 0$ expresa la idea que "falso" es el elemento absorbente de $\wedge$.

$$\begin{array}{rcl} [\![true]\!](E) & = & 1 \\ [\![false]\!](E) & = & 0 \\ [\![\neg bexp]\!](E) & = & \begin{cases} 0 & \text{si } [\![bexp]\!](E) > 0 \\ 1 & \text{si } [\![bexp]\!](E) = 0 \end{cases} \\ [\![bexp_1 \wedge bexp_2]\!](E) & = & [\![bexp_1]\!](E) * [\![bexp_2]\!](E) \\ [\![bexp_1 \vee bexp_2]\!](E) & = & [\![bexp_1]\!](E) + [\![bexp_2]\!](E) \end{array}$$

Ahora, los comandos o sentencias: $[\![cmd]\!](E)$ devuelve el estado modificado ejecutando el comando cmd a partir del estado E. El comando **skip** no hace nada, por eso el nuevo estado es el mismo E.

$$[\![\textsf{skip}]\!](E) = E$$

Asignar el valor de una expresión a una variable significa evaluar la expresión usando las reglas anteriores y luego guardar su valor en la posición correspondiente a x:

$$[\![x := exp]\!](E) = E\,[x \leftarrow ([\![exp]\!](E))]$$

La concatenación de dos comandos se describe aplicando la semántica del segundo comando al estado obtenido tras ejecutar el primer comando:

$$[\![cmd_1;\ cmd_2]\!](E) = [\![cmd_2]\!]([\![cmd_1]\!](E))$$

Una sentencia condicional se ejecuta evaluando la condición y, si es verdadera (es decir, > 0) se ejecuta la rama "then", en caso contrario la rama "else":

$$[\![\textbf{if } bexp \textbf{ then } cmd_1 \textbf{ else } cmd_2]\!](E) = \begin{cases} [\![cmd_1]\!](E) & \text{si } [\![bexp]\!](E) > 0 \\ [\![cmd_2]\!](E) & \text{si } [\![bexp]\!](E) = 0 \end{cases}$$

La semántica de un bucle indica que el cuerpo del bucle se ejecuta sólo si la condición es verdadera y, en este caso, lo siguiente es volver a evaluar la condicón hasta que quede falsa. Es importante notar que en este caso puede haber una recursión infinita porque la semántica de "**while** *bexp* **do** *cmd*" se define en términos de la semántica de "*cmd*; **while** *bexp* **do** *cmd*" que incluye el mismo bucle. Esto es coherente con la idea de que un bucle puede no terminar, y si termina es porque ejecutar el cuerpo modifica el estado hasta que tarde o temprano la condición del bucle se vuelva falsa.

$$[\![\textbf{while } bexp \textbf{ do } cmd]\!](E) = \begin{cases} E & \text{si } [\![bexp]\!](E) = 0 \\ [\![cmd;\ \textbf{while } bexp \textbf{ do } cmd]\!](E) & \text{si } [\![bexp]\!](E) > 0 \end{cases}$$

Aplicar esta semántica a programas concretos no hace otra cosa que confirmar nuestra intuición a la hora de mirar un programa y ver lo que hace: evaluar expresiones, modificar variables, actuar según el valor de una condición, etc. Sin embargo, es útil porque se trata de una definición formal que será el punto de partida para la siguiente sección.

11.2 Aproximaciones de la semántica

El resultado fundamental del Capítulo 9 es que, en general, no se puede decidir si un programa calcula o no cierta función. Sin embargo, la semántica descrita en la Sección 11.1 parece decir lo contrario: basta "ejecutarla" y se tendrá toda la información sobre lo que el programa calcula. El problema es que esta semántica (1) necesita partir de un valor concreto de las variables; y (2) no evita el problema de la terminación: si un bucle no termina, tampoco el cálculo de la semántica termina. De hecho, calcular la semántica es equivalente en todo y por todo a ejecutar el programa, pero esto no cubre todos los inputs y puede dar lugar a computaciones infinitas.

Por lo tanto, la semántica aquí descrita no puede ser usada como analizador estático, porque dos cosas que se piden a un analizador estático son, entre otras, (1) que "cubra" todos los inputs posibles, es decir, que sea capaz de decir algo para todo caso posible de ejecución; y (2) que termine siempre. El planteamiento descrito en este capítulo es uno de los modos posibles

para definir un analizador estático, y consiste en *aproximar* el resultado de la semántica, lo que implica perder precisión (equivocarse algunas veces) pero lleva a un algoritmo que siempre termina.

Esta aproximación depende en primer lugar de la propiedad que se quiere estudiar: por ejemplo, supongamos que lo único que interesa de una variable numérica es (1) si es 0; (2) si es positiva; (3) si es negativa; o (4) una combinación de estas posibilidades. Se trata de una aproximación porque se pierde el valor exacto, pero esta aproximación es suficiente si, por ejemplo, el objetivo es demostrar que un programa nunca ejecuta una división por 0: si en el punto de programa inmediatamente antes de la división se puede demostrar que el denominador nunca es 0 (aunque su valor sea desconocido), entonces queda garantizado que la división no generará ningún error.

11.2.1 Semántica aproximada de las expresiones

La semántica aproximada se basa en un *estado aproximado* A, que para toda variable sólo puede hacer una de las siguientes afirmaciones:

que es 0	(representado con [**0**])
que es positivo	(representado con [**pos**])
que es negativo	(representado con [**neg**])
que es mayor o igual a 0	(representado con [**n-neg**])
que es menor o igual a 0	(representado con [**n-pos**])
que no es 0	(representado con [**n-0**])
que no se sabe nada de él	(representado con [**¿?**])

Ejemplo 11.2 *Sea A tal que $A(\mathsf{x}) = [\mathbf{pos}]$, $A(\mathsf{y}) = [\mathbf{n\text{-}pos}]$, $A(\mathsf{z}) = [\mathbf{0}]$. Este estado aproximado contiene la información que* x *es positiva,* y *es no negativa, y* z *es exactamente* 0*.*

Estos *valores aproximados* en realidad representan conjuntos de valores: por ejemplo, [**pos**] representa todos los números positivos, [**n-neg**] representa todos los números no-negativos (los positivos más 0), [**0**] representa el conjunto $\{0\}$. Un valor aproximado X es una *aproximación correcta* de un número n si n es uno de los valores representados por X; X es la *mejor aproximación* de n si no existe otra aproximación correcta Y de n que represente menos números[3].

[3]La existencia de una única mejor aproximación depende de ciertas propiedades de la aproximaciones que no vamos a tratar aquí.

Ejemplo 11.3 *Los valores aproximados* [**¿?**], [**n-0**], [**n-pos**] *y* [**neg**] *son todos aproximaciones correctas de* -5 *porque se trata de un número que es, al mismo tiempo, entero (es decir, representado por* [**¿?**]*), distinto de* 0 *(es decir, representado por* [**n-0**]*), no-positivo (es decir, representado por* [**n-pos**]*) y negativo (es decir, representado por* [**neg**]*). Sin embargo, la mejor aproximación de* -5 *es* [**neg**] *porque implica (es más "pequeña" que) todas las demás (es decir, todo número representado por* [**neg**] *también lo es por* [**n-pos**], [**n-0**] *y* [**¿?**]*).*

Ejemplo 11.4 *El estado A del Ejemplo 11.2 es una aproximación correcta de los estados concretos* E_1 *y* E_2 *tales que*

$$\begin{array}{lcr} E_1(\mathsf{x}) & = & 3 \\ E_1(\mathsf{y}) & = & -8 \\ E_1(\mathsf{z}) & = & 0 \end{array} \qquad \begin{array}{lcr} E_2(\mathsf{x}) & = & 1 \\ E_2(\mathsf{y}) & = & -1 \\ E_2(\mathsf{z}) & = & 0 \end{array}$$

En cambio, A no aproxima correctamente los estados E_3 *y* E_4*:*

$$\begin{array}{lcr} E_3(\mathsf{x}) & = & 0 \\ E_3(\mathsf{y}) & = & 0 \\ E_3(\mathsf{z}) & = & 0 \end{array} \qquad \begin{array}{lcr} E_4(\mathsf{x}) & = & 1 \\ E_4(\mathsf{y}) & = & -1 \\ E_4(\mathsf{z}) & = & 2 \end{array}$$

Con esta idea de aproximación se puede definir una *aritmética alternativa* que dice cómo se combinan los valores aproximados, y aproxima correctamente la aritmética estándar. Los nuevos operadores se denotan como [+], [−], [∗], y [/]. Por ejemplo:

[**0**] [+] [**pos**] = [**pos**]	(0 más un número positivo es positivo)
[**pos**] [+] [**pos**] = [**pos**]	(la suma de dos positivos es positiva)
[**n-neg**][+][**n-neg**]=[**n-neg**]	(la suma de dos no-negativos es no-negativa)
[**n-neg**][+][**n-pos**]= [**¿?**]	(nada se sabe de la suma de un no-negativo y un no-positivo: puede ser > 0, < 0 o 0)
[**0**] [∗] [**neg**] = [**0**]	(0 por un número negativo es 0)
[**0**] [∗] [**¿?**] = [**0**]	(incluso 0 por un número desconocido es 0)

Ya se puede apreciar una primera ventaja de usar esta aproximación: el número de valores que una variable puede tener ya no es infinito, sino que es un número relativamente pequeño y manejable: 7. Aún así, el número de casos a tener en cuenta en la definición de una operación aritmética o lógica aproximada no es pequeño, pero existen técnicas para optimizarlo.

Ejemplo 11.5 *En principio, la suma aproximada* $[+]$ *se puede definir especificando el resultado para toda pareja de valores (*$[\mathbf{0}]$ *con* $[\mathbf{0}]$, $[\mathbf{0}]$ *con* $[\mathbf{pos}]$, $[\mathbf{0}]$ *con* $[\mathbf{neg}]$*, etc.), lo que daría 49 posibles combinaciones. Esto sucede porque, al tener que ejecutar una suma aproximada, los valores aproximados de los sumandos, calculados anteriormente, podrían ser cualquiera de los 7 especificados arriba. Sin embargo, es fácil observar que sumar* $[\mathbf{¿?}]$ *con cualquier otro valor (incluso cambiando el orden) siempre da como resultado* $[\mathbf{¿?}]$*, porque el desconocimiento total de un sumando no permite decir nada de la suma. Aprovechando estas características del operador, se puede reducir bastante el número de casos a considerar.*

La aritmética aproximada tiene que ser *correcta* con respecto a la aritmética estándar. Esto significa que, dada por ejemplo una expresión n_1+n_2, cogiendo dos valores aproximados X_1 que aproxima correctamente n_1 y X_2 que aproxima correctamente n_2, el resultado $X_1[+]X_2$ es una aproximación correcta de $n_1 + n_2$.

Ejemplo 11.6 *La versión aproximada de la expresión* $-4+7$ *es* $[\mathbf{neg}][+][\mathbf{pos}]$*, y su resultado es* $[\mathbf{¿?}]$*, que representa correctamente el resultado* 3*. Notamos que* $[\mathbf{¿?}]$ *no es la mejor aproximación de* 3 *(de hecho, es la peor), pero esto es el precio que a veces hay que pagar por aproximar la semántica. En cambio, la versión aproximada de* $-4 * 7$ *es* $[\mathbf{neg}][*][\mathbf{pos}]$*, cuyo resultado es* $[\mathbf{neg}]$*, que sí es la mejor aproximación del resultado real* -28*: en este caso, no hay perdida de información más que la debida a la aproximación.*

El objetivo final de esta sección es definir una versión aproximada $[\![\ ]\!]^{\#}$ de $[\![\ ]\!]$ que trabaje con los valores aproximados. El valor devuelto por $[\![n]\!]^{\#}(A)$ donde n es una constante numérica es la mejor aproximación posible de n: $[\mathbf{0}]$ si $n = 0$, $[\mathbf{pos}]$ si $n > 0$, o $[\mathbf{neg}]$ si $n < 0$. Lo dicho acerca de las operaciones aritméticas aproximadas indica cómo definir la parte de $[\![\ ]\!]^{\#}$ que evalúa las expresiones: básicamente, $[+]$ reemplaza $+$, $[*]$ reemplaza $*$, etc.

11.2.2 Semántica aproximada de las sentencias

La mayoría de las reglas de la semántica aproximada $[\![\ ]\!]^{\#}$ que manejan las sentencias son muy parecidas a las correspondientes de $[\![\ ]\!]$, y sólo difieren por la forma de evaluar las expresiones. Los dos casos más complejos son la sentencia condicional y el bucle.

Sentencia condicional. A primera vista, se podría pensar que la regla de $[\![\ ]\!]^\#$ para la sentencia condicional no es muy distinta de la correspondiente regla de $[\![\ ]\!]$, aplicando la semántica aproximada a las dos ramas del condicional. Sin embargo, hay una diferencia importante: en $[\![\ ]\!]$, cada vez que se evalúa la condición *exp* en el estado E, su valor puede ser, según los casos, verdadero o falso. En cambio, $[\![\ ]\!]^\#$ admite la posibilidad de que el valor de una condición sea, por así decir, verdadero y falso *a la vez*. Supongamos que el valor aproximado de una variable booleana x antes de la sentencia condicional sea [**n-neg**]: en este caso, la semántica aplicada a

```
if (x) then cmd1 else cmd2;
```

no sabrá decir si la condición es verdadera o falsa, ya que x podría ser tanto 0 (es decir, falso) como positivo (es decir, verdadero). En otras palabras, la información que se pierde aproximando lleva, en algunos casos, a no saber cómo interpretar ciertas partes de un programa.

La solución es *tener en cuenta ambas posibilidades*, tanto la ejecución de la rama "then" como de la rama "else", y calcular el estado final "uniendo" el estado aproximado derivado de la ejecución de cmd1 con el estado correspondiente a cmd2.

Ejemplo 11.7 *Si se aplica* $[\![\ ]\!]^\#$ *al siguiente código*

```
if (x) then { y := 1 } else { y := −1 };
```

a partir de un estado inicial A *tal que* $A(\mathsf{x}) = [\mathbf{n\text{-}neg}]$*, entonces no se puede saber qué camino va a tomar la ejecución. La única forma de tener en cuenta todos los escenarios es calcular* [**n-0**] *como valor de* y *en el estado después de la sentencia condicional, porque* y *puede ser tanto positiva como negativa, pero, por lo menos, se puede excluir que tome el valor* 0*.*

El nuevo operador $[\cup]$ que une valores aproximados tiene una definición bastante intuitiva:

$$
\begin{array}{lllll}
[\mathbf{pos}] & [\cup] & [\mathbf{pos}] & = & [\mathbf{pos}] \\
[\mathbf{pos}] & [\cup] & [\mathbf{neg}] & = & [\mathbf{n\text{-}0}] \\
[\mathbf{pos}] & [\cup] & [\mathbf{n\text{-}pos}] & = & [\mathbf{¿?}] \\
[\mathbf{pos}] & [\cup] & [\mathbf{n\text{-}neg}] & = & [\mathbf{n\text{-}neg}] \\
[\mathbf{pos}] & [\cup] & [\mathbf{0}] & = & [\mathbf{n\text{-}neg}] \\
[\mathbf{pos}] & [\cup] & [\mathbf{n\text{-}0}] & = & [\mathbf{n\text{-}0}] \\
[\mathbf{pos}] & [\cup] & [\mathbf{¿?}] & = & [\mathbf{¿?}] \qquad \textit{etc.}
\end{array}
$$

El resultado es la mejor aproximación posible de la unión de los valores representados por cada uno de los argumentos.

Sin embargo, hay una posible optimización para alcanzar una aproximación mejor en muchos casos. Se trata de utilizar la información que a veces es proporcionada por la condición de la sentencia condicional: por ejemplo, si esta condición es `i>0`, el análisis de la rama "then" puede garantizar que el valor aproximado de `i` es [**pos**], y el análisis de la rama "else" puede garantizar que [**n-pos**] es una aproximación correcta de `i`. Obviamente, no siempre se puede inferir información útil de la condición: esta discusión se limita a aprovechar casos sencillos como v *op* n, donde v es una variable entera, *op* es un operador de comparación, y n es una constante numérica. La siguiente tabla contiene algunos ejemplos de la información que se puede deducir de una condición booleana de una sentencia condicional al analizar cada una de las dos ramas.

condición	rama "then"	rama "else"
$v > 5$	v es [**pos**]	nada
$v \geq 0$	v es [**n-neg**]	v es [**neg**]
$v \neq -3$	nada	[**neg**]
$v < 1$	v es [**n-pos**]	v es [**pos**]

Ejemplo 11.8 *Sin esta optimización no se puede inferir que el valor final de* `y` *en el siguiente código es no-negativo, porque no se conoce el valor inicial de* `x`.

```
if (x≥0) then { y := x } else { y := −x };
```

Sin embargo, el estudio de la condición booleana asegura que `x` *es positivo o cero en la rama "then", por lo que* `y` *también lo será, y que* `x` *es estrictamente negativo si se ejecuta la rama "else", por lo que* `y` *será estrictamente positivo (*`−x` *es una multiplicación por -1). La unión de las aproximaciones obtenidas en las dos ramas es* [**n-neg**].

El bucle. La semántica aproximada del bucle es un poco más difícil de entender porque es menos intuitiva. Se podría pensar que es suficiente hacer como con el condicional: tener en cuenta ambos caminos y combinar la información al final. La diferencia es que los caminos no son dos, sino potencialmente ilimitados: el cuerpo del bucle se podría ejecutar, en principio, cualquier número no-negativo de veces. Por lo tanto, se necesita algún mecanismo para evitar este problema, y este mecanismo sólo puede ser otra aproximación (es decir, otra perdida de precisión a cambio de una simplificación del problema).

No procede entrar en los detalles del tratamiento de los bucles. Únicamente vamos a decir cómo podría aparecer la solución más sencilla (no necesariamente la mejor): ejecutar el cuerpo *cierto número n de veces* de manera que:

- el efecto de cada ejecución del cuerpo (el estado aproximado) *se une* con el estado anterior, para tener en cuenta todas las ejecuciones posibles; y
- el procedimiento termina cuando ejecutar el bucle una vez más $(n + 1)$ no cambia el estado.

En el caso de la aproximación que se está manejando (la positividad o negatividad de números enteros), estas dos condiciones aseguran que la semántica aproximada ejecuta el cuerpo del bucle un número finito de veces y, por lo tanto, termina. En el caso general, se pueden necesitar otros mecanismos más complejos que aseguren la terminación.

Ejemplo 11.9 *Se considere el siguiente código:*

```
x := 0;
...
while (...) do {
  x := x+1;
  ...
}
```

donde la semántica aproximada no es capaz de decir nada de la condición del bucle ni de lo que pasa en el cuerpo después de la primera línea, pero se sabe que no afecta directamente x*. En este caso resulta posible inferir información útil acerca de* x*.*

El valor aproximado de x *después de la primera línea es* [**0**]*. Al ejecutar el cuerpo del bucle, el valor generado para* x *es* [**pos**]*, porque está garantizado que* x *será positivo. Sin embargo, no se sabe si el cuerpo del bucle se llega o no a ejecutar al menos una vez, así que el valor inicial de* x *no se tiene que perder. Por lo tanto, después de cero o una iteración,* x *vale la unión de* [**0**] *y de* [**pos**]*, es decir,* [**0**][∪][**pos**] = [**n-neg**]*. Lo interesante es que, si se "ejecuta" otra vez el cuerpo del bucle partiendo de* [**n-neg**] *como valor de* x*, el valor que sale, después de calcular la unión, sigue siendo* [**n-neg**]*, y esto no va a cambiar ejecutando más veces el bucle. En otras palabras, si el valor aproximado de todas las variables que se almacenan en el estado no cambia pasando de la iteración* n *a la* $n + 1$*, entonces no va a cambiar de allí en adelante. Por lo tanto, el valor aproximado final de* x *es el valor que asume "en el límite":* [**n-neg**]*. Este resultado es correcto porque* x *será realmente no-negativo al final del bucle.*

11.2.3 El "quid" de la cuestión

Resumiendo algo planteado en estas últimas páginas, las razones por la que tiene sentido introducir una semántica aproximada $[\![\ ]\!]^{\#}$ son básicamente dos:

- En el caso de $[\![\ ]\!]$, el número de valores inciales a considerar para cada variable numérica es infinito (en principio, puede ser cualquier número), mientras que en $[\![\ ]\!]^{\#}$ sólo hay unos pocos valores posibles; por lo tanto, el número de estados aproximados a considerar como estados iniciales es finito.

- La perdida adicional de precisión que se asume en el tratamiento de los bucles consigue un resultado muy importante: la semántica $[\![\ ]\!]^{\#}$ *siempre termina*, aunque el número de iteraciones del bucle sea ilimitado.

Visto desde este punto de vista, que no es el único posible, un analizador estático es un programa que *aplica una semántica aproximada* a otro programa partiendo de un estado aproximado que represente *todo posible input*, y esta semántica está construida a partir de la propiedad que se quiere analizar. Este analizador perderá información con respecto a la ejecución real del programa, con la consecuencia que su resultado puede ser incorrecto. Sin embargo, lo hace a cambio de unas ventajas importantes:

- Puede tener en cuenta efectivamente todos los posibles inputs con un número finito de ejecuciones.

- La ejecución de la semántica aproximada termina para todo programa y para todo estado aproximado inicial.

- Por como se gestiona la perdida de información debida a la aproximación, se puede obtener que los errores de clasificación siempre se den en el mismo sentido (sólo falsas alarmas y nunca excesos de confianza).

Este último resultado viene de cómo se define la semántica aproximada. Si $[\![\ ]\!]^{\#}$ devuelve A como estado aproximado final de cierto programa, A aproxima correctamente todos los estados finales posibles y, además, aproxima también algunos estados que son de hecho imposibles (esto puede generar falsas alarmas). Lo que no debe pasar nunca es que algún estado posible no sea aproximado por A, porque en este caso se podría dar lugar a un exceso de confianza.

Ejemplo 11.10 *Sea el siguiente el código completo del Ejercicio 11.9, donde se conoce que el valor inicial de* y *es no-negativo:*

```
x := 0;
z := y;
while (z≠0) do {
  x := x+1;
  z := z/2; // (división entera)
}
```

La ejecución de este fragmento de código terminará en un estado E tal que $E(\mathsf{x})$ es igual a 1 más el logaritmo (entero) en base 2 del valor de y*. La semántica aproximada no es capaz de detectar esta información, y se limita a decir que en el estado final aproximado A ambas variables son no negativas (partiendo de la hipótesis que* y *es inicialmente no-negativo). La razón por la que $A(\mathsf{x}) = [\textbf{n-neg}]$ se explica en el Ejercicio 11.9.*

El estado final aproximado entonces es tal que $A(\mathsf{x}) = A(\mathsf{y}) = [\textbf{n-neg}]$, y es una aproximación correcta de todos *los posibles estados finales. Por ejemplo, podría ser que $E(\mathsf{x}) = 4$ y $E(\mathsf{y}) = 10$, o que $E(\mathsf{x}) = 6$ y $E(\mathsf{y}) = 40$. Sin embargo, A también aproxima estados como $E(\mathsf{x}) = E(\mathsf{y}) = 5$, que nada tiene que ver con la relación logarítmica entre* x *e* y*.*

11.3 ⟦ ⟧# y el problema de la división por cero

Una semántica aproximada como la descrita en este capítulo se puede usar para investigar la posibilidad de ejecutar divisiones por cero (Sección 10.4.1). Obviamente, en la realidad una semántica aproximada sería mucho más sofisticada, precisa y eficiente que ésta, pero el objetivo de este capítulo es simplemente dar una idea de cómo se puede plantear una metodología de análisis estático.

Hasta ahora la tarea llevada a cabo por la semántica aproximada ⟦ ⟧# ha sido ejecutar un programa para calcular un estado final aproximado que de información sobre todos los posibles valores de las variables al final del programa. Sin embargo, no es difícil extender su definición para que la información proporcionada sea más completa: para toda variable v y todo punto de programa p (o línea de código, o sentencia, o etiqueta), $A_p(v)$ tiene que aproximar correctamente todos los posibles valores que v puede tener en p durante alguna ejecución del programa.

El siguiente paso es utilizar esta información para detectar si algunas divisiones contenidas en el código pueden generar un error a tiempo de ejecución. Si el analizador encuentra una división exp1 / exp2, la operación será considerada segura si el valor aproximado calculado para exp2 es uno entre [**pos**], [**neg**]

o [**n-0**], es decir, uno de los valores que no contienen el cero. Si pasa esto, se puede garantizar que esa división nunca tendrá denominador igual a 0.

Ejemplo 11.11 *En este código*

```
1 if (x=0) then {
2   y := 1;
3 } else {
4   y := -1;
5 }
6 z := w/y;
```

la semántica aproximada $[\![\ ]\!]^{\#}$ *puede calcular el valor* [**n-0**] *para* y *antes de la línea 6, excluyendo por lo tanto que su valor sea* 0. *Esto supone una pérdida de información notable con respecto a la verdadera ejecución del programa (donde* y *sólo puede tener dos valores:* 1 *y* -1*). Sin embargo, la información calculada es lo suficientemente precisa para poder garantizar que la división de la línea 6 no generará ningún error a tiempo de ejecución.*

11.4 Resumen

Resumen 11.1 *El objetivo de una semántica aproximada es representar todas las posibles ejecuciones de un programa. Además, para ser de alguna utilidad, la semántica tendrá que terminar siempre.*

Resumen 11.2 *Debido a la imposibilidad (por el Teorema de Rice) de conseguir estos objetivos, la semántica aproximada tendrá que representar también ejecuciones que en realidad con imposibles.*

Resumen 11.3 *Para cubrir todos los casos y garantizar la terminación, la semántica aproximada tendrá a veces que considerar ambas ramas de una sentencia condicional, y usar algún mecanismo avanzado de punto fijo para tratar los bucles.*

Resumen 11.4 *Si se aplica una semántica aproximada de este tipo a alguna propiedad interesante de un programa, el resultado será tal que podrá haber falsas alarmas pero nunca excesos de confianza.*

11.5 Ejercicios

Ejercicio 11.1 *Aplicando la intuición dada en el Ejemplo 11.5, definir la suma aproximada* $[+]$ *de manera que la definición (1) sea lo más precisa posible (es decir, no diga que el resultado es* $[\textbf{¿?}]$ *si se puede decir algo más preciso); (2) cubra todos los casos (es decir, que esté definida para cualquier valor aproximado de los sumandos); y (3) use un número mínimo de ecuaciones.*

Ejercicio 11.2 *Aplicando una adaptación de la intuición dada en el Ejemplo 11.5, definir el producto aproximado* $[*]$ *de manera que la definición (1) sea lo más precisa posible (es decir, no diga que el resultado es* $[\textbf{¿?}]$ *si se puede decir algo más preciso); (2) cubra todos los casos (es decir, que esté definida para cualquier valor aproximado de los sumandos); y (3) use un número mínimo de ecuaciones.*

Ejercicio 11.3 *Definir la parte de la semántica* $[\![\ ]\!]^{\#}$ *que se ocupa de evaluar expresiones aritméticas.*

Ejercicio 11.4 *Definir la parte de la semántica* $[\![\ ]\!]^{\#}$ *que se ocupa de evaluar expresiones booleanas (teniendo en cuenta que los valores booleanos se pueden representar como números, pero el sistema de tipos usado en este lenguaje asegura que una expresión booleana nunca dará un valor numérico negativo.*

Ejercicio 11.5 *Definir la semántica* $[\![\ ]\!]^{\#}$ *para la sentencia condicional, indicando cómo se puede tener en cuenta ambos caminos en los casos en que no se puede establecer con seguridad el valor de verdad de la condición. Pista: definir un operador* $[\cup]$ *que "una" los valores aproximados de manera que, por ejemplo,* $[\textbf{neg}][\cup][\textbf{pos}] = [\textbf{n-0}]$ *y* $[\textbf{0}][\cup][\textbf{pos}] = [\textbf{n-neg}]$*. ¿A qué operación conocida se parece?*

Ejercicio 11.6 *Sea* P *un programa que usa* 3 *variables numéricas. El analizador* A *aplica a* P *la semántica* $[\![\ ]\!]^{\#}$ *descrita en este capítulo. ¿Cuántos estados aproximados tendrá que considerar como posibles estados iniciales para cubrir todas las posibles ejecuciones de* P*? Nota: es probable que la primera respuesta que viene a la cabeza no sea la correcta. Es más, se podría decir que hay más de una respuesta correcta, según la precisión que se requiere y la cantidad de trabajo de* A *que se está dispuestos a permitir.*

Ejercicio 11.7 *Evaluar la expresión* $3 + (-2) * 5$ *según la semántica* $[\![\ ]\!]^{\#}$*, de la siguiente forma: (1) transformar cada valor numérico en su aproximación, la más precisa posible; (2) aplicar los operadores aproximados* $[+]$*,* $[*]$*, etc.*

Ejercicio 11.8 *Evaluar la expresión* $7^2 + (-6)^2$ *según la semántica* $[\![\ ]\!]^\#$*, de la siguiente forma: (1) transformar cada valor numérico en su aproximación, la más precisa posible; (2) aplicar los operadores aproximados* $[+]$, $[*]$*, etc.*

Ejercicio 11.9 *Evaluar la expresión* $(2-1)*(5-5)$ *según la semántica* $[\![\ ]\!]^\#$*, de la siguiente forma: (1) transformar cada valor numérico en su aproximación, la más precisa posible; (2) aplicar los operadores aproximados* $[+]$, $[*]$*, etc.*

Ejercicio 11.10 *Evaluar la expresión* $x^2 + 5 * 9$*, donde no se sabe nada de* x*, según la semántica* $[\![\ ]\!]^\#$*, de la siguiente forma: (1) transformar cada valor numérico en su aproximación, la más precisa posible; (2) aplicar los operadores aproximados* $[+]$, $[*]$*, etc. En este caso hay que definir otra operación aproximada* $^{[2]}$ *que aproxima la elevación al cuadrado.*

Ejercicio 11.11 *Evaluar la expresión* $[\mathbf{n\text{-}neg}][+]([\mathbf{n\text{-}0}]^{[2]}[+][\mathbf{pos}])$ *según* $[\![\ ]\!]^\#$*.*

Ejercicio 11.12 *Evaluar la expresión* $([\mathbf{n\text{-}neg}][-][\mathbf{n\text{-}pos}])[+][\mathbf{¿?}]^{[2]}$ *según la semántica* $[\![\ ]\!]^\#$*.*

Ejercicio 11.13 *Evaluar la expresión* $[\mathbf{n\text{-}0}][*][\mathbf{neg}]$ *según la semántica* $[\![\ ]\!]^\#$*.*

Ejercicio 11.14 *Evaluar según* $[\![\ ]\!]^\#$ *la expresión aritmética*

$$[\mathbf{pos}][+]([\mathbf{¿?}][*][\mathbf{0}])[-]([\mathbf{n\text{-}pos}][*][\mathbf{n\text{-}0}])$$

Ejercicio 11.15 *Por cómo se definió la semántica del lenguaje, "verdadero" se representa como cualquier número positivo y "falso" es el número* 0*. Así que la semántica aproximada* $[\![\ ]\!]^\#$ *podría interpretar "verdadero" como* $[\mathbf{pos}]$*, "falso" como* $[\mathbf{0}]$ *y "desconocido" como la unión de ambos* $[\mathbf{n\text{-}neg}]$*. Definir entonces el operador "menor o igual"* $[\leq]$*. Por ejemplo:*

$[\mathbf{0}][\leq][\mathbf{0}] = [\mathbf{pos}]$ *(verdadero porque* $0 \leq 0$*)*
$[\mathbf{0}][\leq][\mathbf{pos}] = [\mathbf{pos}]$ *(verdadero porque* $0 \leq n$ *si* n *es positivo)*
$[\mathbf{0}][\leq][\mathbf{n\text{-}0}] = [\mathbf{n\text{-}neg}]$ *(no se sabe: en* $[\mathbf{n\text{-}0}]$ *hay pos. y neg.)*
etc. (completar)

Ejercicio 11.16 *Hacer lo mismo que el Ejercicio 11.15 para los otros operadores de comparación entre números:* $[=]$, $[<]$, $[\geq]$, $[>]$, $[\neq]$*.*

Ejercicio 11.17 *Definir los operadores lógicos* $[\neg]$, $[\vee]$, $[\wedge]$, $[\rightarrow]$ *(implicación lógica) y* $[\leftrightarrow]$ *(doble implicación). Por ejemplo,* $[\neg][\mathbf{pos}] = [\mathbf{0}]$ *porque la negación de "verdadero" es "falso";* $[\mathbf{0}][\wedge][\mathbf{n\text{-}neg}] = [\mathbf{0}]$ *porque la conjunción de "falso" con otra cosa es falsa;* $[\mathbf{pos}][\vee][\mathbf{0}] = [\mathbf{pos}]$ *porque "falso" es el elemento neutro de la disyunción. Completar las definiciones.*

Ejercicio 11.18 *Discutir cómo la optimización que extrae información de la condición booleana de una sentencia condicional puede ser mejorada, yendo más allá del patrón v op n. Escribir una tabla que ejemplifique la información inferida en casos más complejos.*

Ejercicio 11.19 *Discutir cómo la misma optimización usada en el análisis de la sentencia condicional puede ser aplicada al estudio de los bucles.*

Ejercicio 11.20 *Basándose en la definición de* $[\![\ ]\!]^{\#}$*, definir otras tres semánticas aproximadas que usen otros tipos de aproximaciones sobre los valores de las variables. Decir si son más o menos precisas que* $[\![\ ]\!]^{\#}$*, y en cuáles casos. Determinar si la aplicación de estas semánticas aproximadas podría en algunos casos no terminar.*

Ejercicio 11.21 *Para cada una de las semánticas definidas en el Ejercicio 11.20, aplicarlas a un programa sencillo y calcular el estado final aproximado.*

Capítulo 12

Otras consecuencias

Este capítulo termina nuestro recorrido acercándonos (sin ninguna pretensión de ser exhaustivos) a otras consecuencias interesantes de los resultados matemáticos vistos hasta ahora. La primera consecuencia tiene que ver con el análisis de programas, al igual que los capítulos 10 y 11, pero lo que cambia es el punto de vista: en lugar de intentar saber más sobre un programa, lo que se intenta es que *no se pueda saber mucho* sobre él. La segunda consecuencia[1] es uno de los resultados científicos más profundos obtenidos en el siglo XX (y, tratándose del mismo siglo que vio nacer la Teoría de la Relatividad y la Mecánica Quántica, esta afirmación lo dice todo), y establece unos límites a lo que las matemáticas pueden demostrar. La tercera consecuencia es más filosófica, y no todo el mundo está de acuerdo sobre su significado más profundo; tiene que ver con la relación entre lo que puede hacer un hombre y lo que pueden hacer unas máquinas.

12.1 Ofuscación de código

Hay situaciones en las que quien produce el programa tiene interés en que *no* se pueda analizar su código de forma precisa para inferir información útil sobre él. En este caso, lo que se suele hacer es *transformar* el programa P, obteniendo una nueva versión P', de manera que P'

- calcule exactamente lo mismo que P (sea equivalente);

[1] Un lógico matemático hablaría probablemente de "causa" en lugar de "consecuencia". Quizá sea las dos cosas a la vez.

- sea más difícil de analizar de forma precisa.

Este procedimiento se llama *Ofuscación de código* (*Code Obfuscation*) y es un área de investigación bastante activa hoy en día.

Ejemplo 12.1 *Usando los mecanismos y propiedades introducidos en el Capítulo 11, se quiere analizar el siguiente programa con la semántica aproximada* $[\![\ ]\!]^{\#}$*:*

```
1 var i:int, j:int;
2 begin
3   i := 0;
4   while (i<=10) do
5     i := i+1;
6   j := 11/i;
7 end
```

Es fácil ver que el valor de i *antes de ejecutar la línea 6 será* 11 *y nunca* 0*, por lo que la división no generará ningún error a tiempo de ejecución. Sin embargo, la semántica* $[\![\ ]\!]^{\#}$ *actúa de la siguiente forma:*

- *calcula* $[\mathbf{0}]$ *como valor aproximado de* i *tras la línea 3;*
- *según el Ejercicio 11.15, el operador* $[\leq]$ *que aproxima* <= *se puede definir correctamente de modo que* $[\mathbf{0}][\leq][\mathbf{pos}]$ *(donde* $[\mathbf{pos}]$ *es la aproximación de la constante* 10*) es verdadero, es decir,* $[\mathbf{pos}]$*;*
- *por lo tanto, la semántica puede garantizar que* por lo menos una *iteración del bucle se va a ejecutar, así que el caso "ninguna iteración" no tiene que ser tenido en cuenta;*
- *ejecutar el incremento de* i *en el estado* A *tal que* $A(\mathsf{i}) = [\mathbf{0}]$ *da* $[\mathbf{pos}]$ *como nuevo valor de* i *(no hay que unirlo con el valor inicial por lo que acabamos de decir: por lo menos se ejecuta una iteración);*
- *ninguna de la siguientes iteraciones del bucle cambia el valor aproximado de* i*.*

De este modo, $[\![\ ]\!]^{\#}$ *es capaz de deducir que* i *es estrictamente positiva al final del bucle, por lo que puede garantizar que la división de la línea 6 no da error.*

En cambio, se considere otra versión del mismo algoritmo:

```
1 var i:int, j:int;
2 begin
```

```
3   i := 0;
4   while (i<=10) do
5     if (i>100) then i := i-1;
6     else i := i+1;
7   j := 11/i;
8 end
```

*La sentencia condicional que se ha introducido es especial porque nunca se ejecuta la rama "then" (*i *no puede ser mayor que* 100 *si es menor o igual a* 10*), así que, en realidad, lo que calcula esta versión del algoritmo es* exactamente lo mismo *que la versión original. Sin embargo,* $[\![\ ]\!]^{\#}$ no *es capaz de darse cuenta de que siempre se ejecuta la misma rama porque la condición* i>100 *se aproxima con* [**pos**][>][**pos**]*, que es algo que puede ser tanto verdadero como falso*[2] *(por lo que viene a ser* [**n-neg**]*). Por lo tanto, en el análisis estático no queda más remedio que considerar ambas ramas de la sentencia condicional, y el problema es que la rama "then" pierde cualquier información (aproximada) sobre* i *porque* [**pos**][−][**pos**] = [**¿?**]*. La conclusión es que el analizador no puede garantizar que la segunda versión del algoritmo es libre de errores porque el valor de* i *después del bucle es* [**¿?**] *y no se puede garantizar la ausencia de divisiones con denominador* 0*.*

Este ejemplo no es realista, y un analizador estático real no se dejaría engañar tan fácilmente por una transformación tan sencilla del programa original. Además, los análisis que se implementan y se usan para realizar este tipo de tareas de control se basan en propiedades más complejas que simplemente "positivo, negativo o cero", y cuentan con una serie de optimizaciones que permitirían tratar de forma correcta este ejemplo (es decir, detectar que no hay divisiones con denominador 0 en ninguno de los dos programas). Sin embargo, el Ejemplo 12.1 da la intuición de lo que pasa: partiendo de un programa que puede ser analizado con precisión por un analizador, siempre existe otro programa equivalente por el que el resultado del mismo analizador es menos preciso. Se trata de otra consecuencia del Teorema de Rice, ya que el teorema demuestra que las propiedades semánticas no triviales son indecidibles.

Más precisamente, la propiedad semántica de la que se habla en este caso es $I = \{x \mid \forall i, j.\varphi_x(i, j)>0\}$, es decir, los índices de las funciones calculables totales que siempre devuelven un número positivo. Para ver que el analizador no decide I, se cogen dos versiones de los programas del Ejemplo 12.1 donde se

[2]Incluso teniendo en cuenta que el valor de i dentro del bucle no puede ser más de 10, como hace normalmente un analizador real, no se llega a inferir nada útil sobre el valor de verdad de la condición.

ha reemplazado la división por un **return** i, ya que en realidad sólo no interesa el valor de i después del bucle, no el resultado de la división.

```
1 var i:int, j:int;
2 begin
3   i := 0;
4   while (i<=10) do
5     i := i+1;
6   return i;
7 end
```

```
1 var i:int, j:int;
2 begin
3   i := 0;
4   while (i<=10) do
5     if (i>100) then i := i-1;
6     else i := i+1;
7   return i;
8 end
```

El índice de ambos programas pertenece a I porque el valor de retorno es positivo para todo input, pero el analizador no puede clasificar correctamente el segundo programa porque el valor aproximado que calcula para i es [¿?].

Un programa P' que calcula lo mismo que otro programa P, pero es más difícil de analizar, se llama *versión ofuscada* de P. La *ofuscación de código* (*code obfuscation*) es una técnica que aplica *tranformaciones* a programas para que sean más difíciles de analizar pero sean equivalentes a los originales desde el punto de vista de la función calculada.

Definición 12.1 *Una* transformación de programas *(*program transformation*) es una función parcial*[3] *t de programas a programas tal que, dado un programa P, él y su transformación $P' = t(p)$ calculan la misma función. En este capítulo se usan transformaciones tales que*

- *tanto P como $t(P)$ son programas* WHILE*; y*
- *el objetivo de la transformación es que sea más difícil para el analizador estático usado analizar $t(P)$ con respecto a P.*

Hecho 12.1 *Dado un programa, se puede transformarlo (ofuscarlo) de modo que siga calculando la misma función, pero su análisis sea más difícil.*

Ejemplo 12.2 *Un compilador es un transformador de programas: la transformación que aplica recibe como input un programa P escrito en un lenguaje L y devuelve un programa $t(P)$ escrito en otro lenguaje L', y ambos programas calculan la misma función. Cualquier herramienta de optimización de código es un transformador de programas: la eliminación de código muerto, operaciones inútiles, variables sin utilizar son transformaciones de programas.*

[3] La transformación no es total porque no tiene por qué aplicarse a programas incorrectos, como sucede por ejemplo en el caso del compilador del Ejemplo 12.2.

En este capítulo se usa una transformación bastante sencilla, llamada *introducción de predicados opacos*. Un *predicado opaco* es una condición booleana B que

- tiene siempre el mismo valor de verdad (es siempre verdadera o siempre falsa); y, sin embargo,
- no puede ser identificada por el analizador ni como verdadera ni como falsa.

Por lo tanto, se crea un desnivel entre lo que B es y lo que el analizador puede decir acerca de ella. Si B es la condición de una sentencia condicional, este desnivel hace que el programa P' siempre ejecute la misma rama de la sentencia, pero el analizador tenga que considerar la posible ejecución de ambas.

Definición 12.2 *Sea P un programa y A un analizador estático. Una transformación t_A de introducción de predicados opacos selecciona un comando*[4] *C que aparece en P, y devuelve un programa P' tal que*

- *C' es otro comando que hace algo* significativamente distinto *con respecto a C, es decir, A calculará dos estados finales aproximados distintos en los dos casos;*
- *B es una condición booleana que comple las condiciones que acaban de ser enunciadas: que siempre tiene el mismo valor de verdad, y que A es incapaz de determinarlo;*
- *si B es siempre verdadera, entonces P' se obtiene de P reemplazando C por el comando* **if** (B) **then** C **else** C'*;*
- *si B es siempre falsa, entonces P' se obtiene de P reemplazando C por el comando* **if** (B) **then** C' **else** C*;*

El programa resultante es equivalente al original desde el punto de vista de la función calculada, porque C' nunca se ejecuta. Sin embargo, el resultado de A será menos preciso porque el analizador tendrá que unir la información correspondiente a la ejecución de C con la correspondiente a C'.

[4]En realidad una transformación podría actuar sobre más de un comando, pero esto no cambia las características del problema, por lo que nos limitamos a un único comando afectado.

Ejemplo 12.3 *El segundo programa del Ejemplo 12.1 se obtiene a partir del primero aplicando una transformación de introducción de predicados opacos. En este caso, el comando C afectado es* `i := i+1`*, la condición booleana introducida es* `i<100` *y el comando alternativo C' es* `i := i−1`*. La ofuscación es efectiva con respecto a $[\![\ ]\!]^{\#}$ porque*

- *B es siempre falsa;*
- *$[\![\ ]\!]^{\#}$ no puede determinar el valor de verdad de B; y*
- *dado el contexto, C' calcula algo significativamente distinto con respecto a C: C siempre asigna un valor estrictamente positivo a* `i` *si ya lo es, mientras que C' no puede garantizarlo.*

La efectividad de una ofuscación de código depende de lo difícil que es analizar el programa ofuscado on respecto al original, y de otros medidores de calidad como la eficiencia o el tamaño de P' con respecto a P y el tiempo necesario para aplicar la ofuscación.

Los siguientes párrafos presentan brevemente dos aplicaciones de la ofuscación de código.

Protección de la propiedad intelectual. Una primera aplicación de la ofuscación de código es la *protección* del software: los productores de software que venden sus programas tienen interés en que no sea posible robar, modificar o sabotear (por ejemplo quitando una *marca de agua digital* o *watermark* que testifique la propiedad del software) el código por parte de hackers que busquen ilegalmente algún tipo de beneficio. En particular, lo que normalmente se busca es que no sea posible hacer *reverse engineering* para obtener el código fuente y acceder de este modo a la estructura original del programa, lo que permitiría modificarlo o sabotearlo bastante fácilmente.

Al fin y al cabo, se puede ver el reverse engineering como una tarea de análisis de programas, por lo que tiene sentido plantearse el diseño de técnicas que lo hagan más difícil y menos efectivo. La ofuscación de código es una de estas técnicas, y puede hacer que el reverse engineering sea imposible o mucho más difícil.

Una ofuscación puede ser tan potente que puede incluso reemplazar el uso de *smartcards* físicas para proteger contenidos como el acceso a unos canales de pago. Ya existen empresa que ofrecen este servicio: la seguridad ya no es a nivel de *hardware* (una tarjeta física que el cliente introduce en un dispositivo), sino de *software*.

Además, programas como Skype usan ofuscación de código (junto con muchas otras técnicas afines) para proteger su contenido[1] evitando robos de código e intrusiones en el protocolo de comunicación. El revés de la medalla, desde el punto de vista de un administrador de sistema, es que también es más difícil detectar ataques como troyanos, malware, etc.

Malware. También los "malos" de la película pueden tener interés en ofuscar su código: es el caso de los que producen algún tipo de *malware*. En este caso, el analizador que pretende extraer información de un programa es el antivirus u otro sistema de seguridad. Las técnicas usadas en el diseño de malware para impedir que los antivirus detecten programas peligrosos son de lo más interesante que hay en el campo de la Ingeniería del Software:

- *predicados opacos*;
- *código repetido*: replicando el código se genera confusión;
- *código desordenado*: operaciones que están semánticamente relacionadas se encuentran en zonas muy distantes del código;
- sustitución de expresiones sencillas por expresiones equivalentes más complejas (por ejemplo sustituir `x=42*y` por `x=y<<5; x+=y<<3; x+=y <<1`, donde `<<` es el operador de *shift*);
- *código mutante*: programas que se auto-modifican durante la ejecución;
- un largo etc.

12.2 Los Teoremas de de Gödel

En los años 30 del Siglo XX, Kurt Gödel demostró dos *Teoremas de Incompletitud* que minaron las fundamentas de las matemáticas (ya se habló de ellos en la Sección 1.2). El primer teorema afirma que en un sistema formal no trivial (es decir, que incluya al menos los axiomas de la aritmética y en el que se puedan demostrar teoremas interesantes de aritmética) hay proposiciones lógicas (afirmaciones o enunciados) que *son verdaderas y sin embargo no se pueden demostrar* dentro del mismo sistema. Este resultado fue un golpe mortal para la corriente racionalista encabezada en ese momento por David Hilbert, que pretendía encontrar un método sistemático para *demostrar todo lo verdadero*. El segundo teorema profundiza el resultado del primero y dice que una de

estas proposiciones indemostrables es "las mátematicas son consistentes". Es decir, no se puede y nunca se podrá demostrar que el método deductivo usado para demostrar un teorema es correcto y sin fallos. Por lo tanto, todo lo que un matemático puede demostrar deductivamente como teorema lo podemos considerar como *absolutamente cierto* sólo si nos olvidamos de Gödel y asumimos que nuestro sistema deductivo no tiene fallos. Sin embargo, el segundo Teorema de Incompletitud nos advierte de que esta pretendida ausencia de fallos *no está demostrada*, y nunca se podrá demostrar. Parece que la certeza absoluta no se puede alcanzar ni siquiera en un ámbito tan sólido como las matemáticas.

La relación entre los Teoremas de Incompletitud y la Teoría de la Computabilidad es muy profunda pero a la vez muy sencilla: la existencia de proposiciones lógicas no demostrables y de problemas no decidibles son las consecuencias de un mismo hecho en la Lógica Matemática y la Informática, respectivamente. Si no existieran problemas indecidibles, tampoco existirían proposiciones lógicas no demostrables, y todo se pudiese demostrar, también existiría un programa para decidir cualquier conjunto de números.

Hecho 12.2 *Existe una relación muy estrecha entre los Teoremas de Incompletitud de Gödel y la Teoría de la Computabilidad: si no hubiera proposiciones lógicas verdaderas pero no demostrables, tampoco habría problemas indecidibles, y viceversa.*

12.3 Problema Mente-Máquina

Una pregunta recurrente en el ámbito de la reflexión filosófica sobre la Inteligencia Artificial es

¿llegarán las máquinas a hacer todo lo que un hombre puede hacer?

Ya se acepta que una máquina ejecute cálculos mucho más rápidamente que un hombre, y que incluso pueda ganar un partido de ajedrez contra el mejor jugador del mundo. Sin embargo, existen tareas donde la capacidad de las máquinas no ha alcanzado todavía el nivel del hombre: traducir un texto, escribir una novela, pintar un cuadro con un valor artístico, llevar a cabo reflexiones filosóficas, etc. Para algunos, sólo es cuestión de tiempo: en el momento en que nuestra comprensión del cerebro sea más profunda, y la potencia de cálculo de un ordenador sea adecuada, no quedará ninguna tarea que no sea accesible a un artefacto mecánico inteligente. Sin embargo, también existen

opiniones discordantes, y una de ellas se apoya justamente en los Teoremas de Gödel.

El matemático y físico Roger Penrose argumenta que los Teoremas de Incompletitud ponen de manifiesto que hay una distancia insanable entre hombre y máquina. La proposición lógica p no demostrable que Gödel usa en el primer Teorema afirma algo como "yo no puedo ser demostrada". A través de un razonamiento que tiene algo que ver con el razonamiento diagonal, se puede ver que

- si p fuera demostrable, entonces sería falsa, ya que p afirma justamente que no se puede demostrar; entonces, p no puede ser demostrable porque si lo fuera el sistema formal que la produce sería inconsistente (en un sistema formal consistente no se puede bajo ningún concepto demostrar algo falso);
- por el mismo hecho de no ser demostrable, p es *verdadera*, ya que lo que dice es cierto.

En resumidas cuentas, lo que no se puede afirmar deductivamente (con una demostración dentro de un sistema formal) sí se puede afirmar fuera del sistema. Ahora, una máquina tal y como la conocemos siempre estará "implementando" algún sistema formal, por lo que siempre existirá una proposición p que no podrá demostrar. Sin embargo, con el razonamiento que se acaba de llevar a cabo, el hombre puede llegar a una certeza sobre esta proposición p.

Hecho 12.3 *Los Teoremas de Gödel son el punto de partida de la opinión de Roger Penrose, que afirma que las máquinas nunca llegarán a hacer todo lo que un hombre hace.*

La teoría de Penrose [16] es fascinante pero demasiado compleja y profunda para ser tratada en este texto. Muchos adversarios de esta teoría no consideran los resultados de incompletitud un factor suficiente para poder afirmar que las máquinas nunca podrán hacer todo lo que un hombre hace. Uno de los argumentos en contra de Penrose es, por ejemplo, que no se puede dar por hecho que una máquina no pueda "salir" de cierto sistema formal.

12.4 Resumen

Resumen 12.1 *Dado un programa, se puede transformarlo de modo que siga calculando la misma función, pero su análisis sea más difícil. Esta transformación se llama Obfuscación de Código.*

Resumen 12.2 *Una técnica sencilla de Ofuscación de Código es insertar una sentencia condicional en la que una rama nunca se ejecuta, pero la condición booleana no puede ser analizada de forma precisa por un analizador, por lo que ambas ramas tienen que ser consideradas como ejecuciones posibles.*

Resumen 12.3 *La Ofuscación de Código se usa para proteger la propiedad intelectual y garantizar la seguridad de un sistema informático.*

Resumen 12.4 *Visto desde el otro punto de vista, la ofuscación puede ser usada para que programas malware no sean detectados por un antivirus u otro mecanismo de seguridad.*

Resumen 12.5 *Existe una relación muy estrecha entre los Teoremas de Incompletitud de Gödel y la Teoría de la Computabilidad: si no hubiera proposiciones lógicas no demostrables, tampoco habría problemas indecidibles, y viceversa.*

Resumen 12.6 *Los Teoremas de Gödel son el punto de partida de la opinión de Roger Penrose, que afirma que las máquinas nunca llegarán a hacer todo lo que un hombre puede hacer.*

12.5 Ejercicios

En los siguientes ejercicios, sea $[\![\ ]\!]^{\#}$ la semántica aproximada del capítulo 11, sin optimizaciones. Sea $[\![\ ]\!]^{\#'}$ la semántica aproximada con la optimización sobre las condiciones booleanas limitada a la sentencia condicional y el caso $v\ op\ n$ (Sección 11.2.2). Sea $[\![\ ]\!]^{\#''}$ la semántica aproximada $[\![\ ]\!]^{\#'}$ refinada tal y como pide el Ejercicio 11.18. Finalmente, sea $[\![\ ]\!]^{\#''}_{w}$ la semántica aproximada $[\![\ ]\!]^{\#''}$ donde el estudio de las condiciones booleanas es aplicado también a los bucles (Ejercicio 11.19).

Ejercicio 12.1 *Aplicar formal y precisamente la semántica $[\![\ ]\!]^{\#}$ a los dos programas del Ejemplo 12.1, y comprobar que efectivamente el resultado del análisis estático no es el mismo en ambos casos.*

Ejercicio 12.2 *Dada la siguiente rutina*

```
1  proc p(i:int) returns int {
2    var j:int, k:int;
3    j := 1;
```

```
4       if (i>0) then {
5         k := i;
6       } else {
7         k := j-i;
8       }
9       return k;
10    }
```

aplicar una transformación de introducción de predicados opacos para que no sea posible para $[\![\ ]\!]^{\#'}$ *determinar que el valor devuelto es estrictamente positivo.*

Ejercicio 12.3 *Dada la siguiente rutina*

```
1   proc p(i:int) returns int {
2     var j:int;
3     j := 1;
4     while (i>0) do {
5       j := j*2;
6       i := i-1;
7     };
8     return j;
9   }
```

aplicar una transformación de introducción de predicados opacos para que no sea posible para $[\![\ ]\!]^{\#'}$ *determinar que el valor devuelto es estrictamente positivo.*

Ejercicio 12.4 *Dada la siguiente rutina*

```
1   proc p(i:int) returns int {
2     var j:int;
3     j := 1;
4     while (i>0) do {
5       j := j*i;
6       i := i-1;
7     };
8     return j;
9   }
```

aplicar una transformación de introducción de predicados opacos para que no sea posible para $[\![\ ]\!]^{\#''}_{w}$ *determinar que el valor devuelto es estrictamente positivo (se note que* `p` *ya está ofuscado para las otras semánticas aproximadas).*

Ejercicio 12.5 *Se considere la siguiente rutina P, donde el valor de input de* x *es positivo.*

```
1  proc p(x:int) returns int {
2    return 2;
3  }
```

La propiedad que se quiere ofuscar es la positividad del valor de retorno. Definir, si es posible, unas transformaciones de introducción de predicados opacos t, t' y t_3 tales que

- *$t(p)$ sea ofuscado (es decir, no permita que el análisis infiera un valor positivo para el valor de retorno) para $[\![\ ]\!]^{\#}$, pero no para las otras semánticas;*
- *$t'(p)$ sea ofuscado para $[\![\ ]\!]^{\#'}$, pero no para $[\![\ ]\!]^{\#''}$ ni $[\![\ ]\!]^{\#''}_w$;*
- *$t''(p)$ sea ofuscado para $[\![\ ]\!]^{\#''}$, pero no para $[\![\ ]\!]^{\#''}_w$;*
- *$t''_w(p)$ sea ofuscado para todas las semánticas aproximadas, incluyendo $[\![\ ]\!]^{\#''}_w$.*

Cada una de las transformaciones puede introducir, si es necesario, más de un predicado opaco (es decir, puede ser la composición de varias transformaciones sencillas).

Ejercicio 12.6 *Dada la siguiente rutina*

```
1  proc p(x:int) returns int {
2    var y:int;
3    y := 1;
4    while (x>0) do {
5      y := y+1;
6      x := x/2; // div. entera
7    }
8    return y;
9  }
```

hacer lo mismo (si es posible) que se pide en el Ejercicio 12.5, teniendo en cuenta que el resultado de la división entera entre dos números positivios en no-negativo.

Ejercicio 12.7 *Dar un ejemplo de rutina tal que:*

- *el valor (no aproximado) devuelto sea siempre negativo;*
- *la semántica* $[\![\ ]\!]^{\#}$ *sabe detectar que el valor devuelto es negativo;*
- *existe una transformación de introducción de uno o más predicados opacos* t *que ofusca la rutina con respecto a* $[\![\ ]\!]^{\#}$*, pero no con respecto a* $[\![\ ]\!]^{\#'}$*,* $[\![\ ]\!]^{\#''}$ *y* $[\![\ ]\!]^{\#''}_{w}$*;*
- *existe una transformación de introducción de uno o más predicados opacos* t'' *que ofusca la rutina con respecto a* $[\![\ ]\!]^{\#}$*,* $[\![\ ]\!]^{\#'}$ *y* $[\![\ ]\!]^{\#''}$*, pero no con respecto a* $[\![\ ]\!]^{\#''}_{w}$*;*
- *existe una transformación de introducción de uno o más predicados opacos* t_w *que ofusca la rutina con respecto a* $[\![\ ]\!]^{\#}$*,* $[\![\ ]\!]^{\#'}$*,* $[\![\ ]\!]^{\#''}$ *y* $[\![\ ]\!]^{\#''}_{w}$*.*

Decir cómo actúan estas transformaciones.

Ejercicio 12.8 *Definir una versión de la transformación de introducción de predicados opacos que se aplique a bucles en lugar de sentencias condicionales.*

Ejercicio 12.9 *Dada la siguiente rutina*

```
1  proc p(x:int) returns int {
2    var y:int, z:int;
3    y := 0;
4    z := 1;
5    while (y<x) do {
6      z := z*4;
7      if (x>5) then {
8        y := y+2;
9      } else {
10       y := y+1;
11     }
12   }
13   return y;
14 }
```

hacer lo mismo (si es posible) que se pide en el Ejercicio 12.5, usando la transformación planteada en el Ejercicio 12.8.

Ejercicio 12.10 *Partiendo de la idea que está detrás de la transformación de introducción de predicados opacos, definir al menos otras tres transformaciones sencillas que sirvan para ofuscar código, bien con respecto a la semántica aproximada del Capítulo 11, bien con respecto a otras semánticas aproximadas. (Sugerencia: fijarse, entre otras cosas, en las operaciones aritméticas).*

Ejercicio 12.11 *Para cada una de las semánticas aproximadas definidas en el Ejercicio 11.20, definir una rutina y una transformación de introducción de predicados opacos que la ofusque de manera significativa.*

Capítulo 13

Bibliografía

[1] Philippe Biondi and Fabrice Desclaux. Silver needle in the skype. Presentado en BlackHat Europe, March 2nd and 3rd, 2006.

[2] George S. Boolos and Richard C. Jeffrey. *Computability and Logic (Second Edition)*. Cambridge University Press, 1980.

[3] Alonzo Church. A Note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1), 1936.

[4] S. Barry Cooper. *Computability Theory*. Chapman & Hall/CRC, 2004.

[5] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[6] Nigel J. Cutland. *Computability: An introduction to recursive function theory*. Cambridge University Press, 1980.

[7] Martin Davis. *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*. Dover Publications, Incorporated, 2004.

[8] H.B. Enderton. In memoriam: Alonzo Church (1903-1995). *Bulletin of Symbolic Logic*, 1(4):486–488, 1995.

[9] Manuel Garrido, editor. *Kurt Gödel: Sobre proposiciones formalmente indecidibles de los Principia Mathematica y sistemas afines.* KRK Ediciones, 2006.

[10] Douglas R. Hofstadter. *Gödel, Escher, Bach: un eterno y gracil bucle.* Tusquets, 2007.

[11] John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introducción a la teoría de Autómatas, Languajes y Computación.* Perason Addison-Wesley, second edition, 2002.

[12] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall, second edition, 1998.

[13] Ernest Nagel, James R. Newman, Kurt Gödel, and Jean-Yves Girard. *Le théorème de Gödel.* Éditions du Seuil, 1997.

[14] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of program analysis.* Springer, 2005.

[15] Piergiorgio Odifreddi. Classical Recursion Theory. *Studies in Logic and the Foundations of Mathematics*, 125, 1992.

[16] Roger Penrose. *The Emperor's New Mind.* Oxford University Press, 1989.

[17] Michael Prasse and Peter Rittgen. Why Church's Thesis Still Holds - Some Notes on Peter Wegner's Tracts on Interaction and Computability. *Arbeitsberichte des Instituts für Wirtschaftsinformatik*, 8, 1997.

[18] Hartley Rogers Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1987.

[19] Herbert A. Simon. Explaining the ineffable. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, 1995.

[20] Alan M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42, 1936.

[21] Alan M. Turing. Computing Machinery and Intelligence. *Mind*, 1950.

[22] Peter Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91.

[23] Peter Wegner and Dina Goldin. Interaction, Computability, and Church's Thesis. 1999.

Anexo A

Soluciones de ejercicios selectos

Capítulo 3

Solución del Ejercicio 3.2. M_1 calcula el doble de un número escrito en notación unaria. Su función de transición se puede escribir como (usando los símbolos auxiliares x e y, y suponiendo que el input sea sintácticamente correcto, es decir, una secuencia de $n+1$ a):

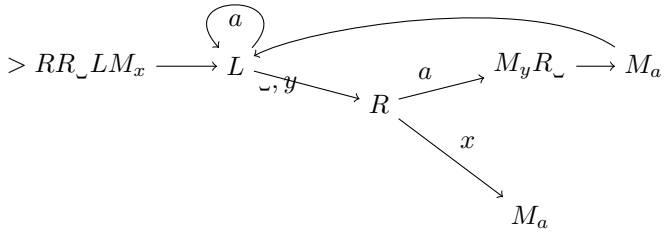
estado	símbolo	nuevo estado	acción	
s	$\sqcup$	q_1	$\rightarrow$	
q_1	a	q_1	$\rightarrow$	busca el final
q_1	$\sqcup$	q_2	$\leftarrow$	
q_2	a	q_3	x	marca el último a
q_3	x	q_3	$\leftarrow$	
q_3	a	q_3	$\leftarrow$	vuelve al principio
q_3	$\sqcup$	q_4	$\rightarrow$	
q_3	y	q_4	$\rightarrow$	
q_4	a	q_5	y	marca el primer a
q_4	x	q_6	a	no hay más a
q_5	y	q_5	$\rightarrow$	
q_5	a	q_5	$\rightarrow$	
q_5	x	q_5	$\rightarrow$	
q_5	$\sqcup$	q_3	a	escribe al final
q_6	a	q_6	$\leftarrow$	
q_6	y	q_6	a	vuelve a poner a
q_6	$\sqcup$	h	$\sqcup$	termina

En cambio M_2 usa la notación binaria, lo que hace su trabajo mucho más fácil (por lo menos en el caso de multiplicar un número por 2):

estado	símbolo	nuevo estado	acción	
s	$\sqcup$	q_1	$\rightarrow$	
q_1	0	q_1	$\rightarrow$	
q_1	1	q_1	$\rightarrow$	
q_1	$\sqcup$	q_2	0	escribe 0 al final
q_2	0	q_2	$\leftarrow$	
q_2	1	q_2	$\leftarrow$	
q_2	$\sqcup$	h	$\leftarrow$	termina

□

Solución del Ejercicio 3.5. El diagrama de M_1 es

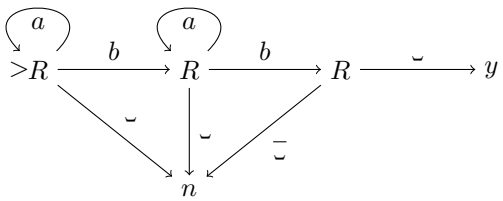


mientras el de M_2 es simplemente

$$> RR␣M_0L␣$$

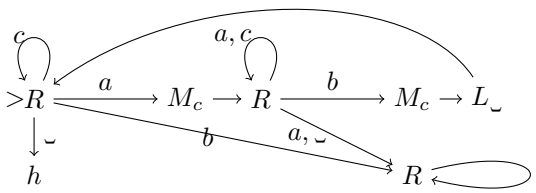
□

Solución del Ejercicio 3.9.



□

Solución del Ejercicio 3.12.



□

Capítulo 4

Solución del Ejercicio 4.3.

$$\begin{cases} pred\,(0) &= const_{0,0} &= 0 \\ pred\,(n+1) &= id_{1,2}(n, pred\,(n)) &= n \end{cases}$$

□

Solución del Ejercicio 4.4. La función $exp\,(n,m) = n^m$ es primitiva recursiva al poder definirse a partir del producto: $exp\,(n,m) = g(m,n)$ y

$$\begin{cases} g(0,n) &= const_{1,1}(n) &= 1 \\ g(m+1,n) &= mul\,(n, g(m,n)) \end{cases}$$

□

Solución del Ejercicio 4.5.

$$\begin{cases} isZero\,(0) &= const_{1,0} &= 1 \\ isZero\,(n+1) &= const_{0,2}(n, isZero\,(n)) &= 0 \end{cases}$$

□

Solución del Ejercicio 4.6. Es fácil darse cuenta de que f es la función constante 0: por ejemplo

$$f(7) = g(6) = f(5) = g(4) = f(3) = g(2) = f(1) = g(0) = 0$$

y esto vale para todo input. Por lo tanto, f es primitiva recursiva. □

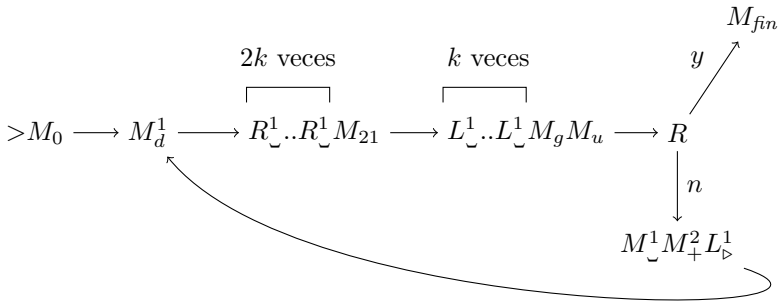
Solución del Ejercicio 4.12. Este ejercicio pide demostrar que, dada una MT M_g que calcule una función g de $k+1$ argumentos, es posible definir otra M_f que implementa f definida como

$$f(n_1,..,n_k) = \begin{cases} \text{el mínimo } m \text{ tal que } g(n_1,..,n_k,m) = 1 & \text{si existe} \\ indefinido & \text{si no existe} \end{cases}$$

La máquina M_g tiene una cinta y, como siempre, encuentra en la cinta los $k+1$ valores de input y, al final de la computación, los habrá borrado todos y escrito el valor $g(n_1,..,n_k,m)$ en la cinta. Además, M_f tendrá dos cintas, y hay que definir algunas máquinas de apoyo:

- M_d duplica el contenido de la cinta (es decir, pasa de $(s, \triangleright_{\lrcorner} w)$ a $(s, \triangleright_{\lrcorner} ww)$);
- M_0 escribe 0 en la cinta;
- M_+ incrementa en 1 su input, y lo deja escrito en el mismo sitio (sobreescribe el input);
- M_u decide si su input es 1, escribiendo en la cinta los símbolo y o n;
- M_{21} copia un número de la segunda cinta al final de la primera.

M_g trabaja en la primera cinta (indicado con M_g^1). En el diagrama, M^k indica que la MT M trabaja en la cinta k. Se supone que todas las máquinas que no sean R o L temrminan su computación en la casilla precedente a donde empezaba su input, como de costumbre. El resultado es (se omiten algunos detalles):



donde M_{fin} es la MT que (1) borra el contenido de ambas cintas manteniendo sólo el valor actual de m (input $k+1$ de g) al principio de la primera cinta; y (2) termina. Las MTs $R_{\lrcorner}$ y $L_{\lrcorner}$ se pueden repetir $2k$ y k veces en el diagrama, respectivamente, porque k es un número fijado a priori, no un input de M_f. □

Capítulo 5

Solución del Ejercicio 5.3. El siguiente programa implementa f:

```
1 funF(n1,..,nk) {
2   ret := funG(n2,..,nk);
3   for i=0 to m-1 do ret := funH(i,ret,n2,..,nk);
4   return ret;
5 }
```

□

Solución del Ejercicio 5.4. La función primitiva recursiva f calcula lo mismo que el programa "for" dado:

$$\begin{cases} f(0, x_2, .., x_k) & = & const_{0,k-1}(0, .., 0) \\ f(x_1{+}1, x_2, .., x_k) & = & sum\,(g(x_1, .., x_k), f(x_1, .., x_k)) \end{cases}$$

□

Solución del Ejercicio 5.7. El siguiente programa implementa la misma función que f:

```
1 funF(n1,...,nk) {
2   i := 0;
3   while (¬ funG(n1,...,nk,i) = 1) i := i+1;
4   return i;
5 }
```

□

Solución del Ejercicio 5.8. La siguiente función μ-Recursiva f calcula lo mismo que funF:

$$\begin{array}{rcl} f_0(0, x_1, .., x_k) & = & const_{0,k}(0, .., 0) = 0 \\ f_0(m+1, x_1, .., x_k) & = & sum\,(h(m, x_1, .., x_k), f_0(m, x_1, x_k)) \\ f(x_1, .., x_k) & = & f_0(\mu y[g(x_1, .., x_k, y)], x_1, .., x_k) \end{array}$$

Lo que hace esta función es calcular por primera cosa el número de iteraciones del bucle con la minimización, y luego ejecutar el cuerpo el número de veces correspondiente. Se podría pensar que hay una discrepancia entre f y funF, ya que la primera podría no llegar nunca a ejecutar el cuerpo del bucle. Sin embargo, si como en este caso sólo nos interesa el resultado final, observamos que si la minimización en la definición de f no termina, entonces el **while** tampoco termina, mientras que si la minimización termina, también termina el bucle. En otras palabras, no importa si una computación que no termina ha ejecutado 1 o 100 veces el cuerpo del bucle, lo importante es que no termina, es decir, no devuelve ningún resultado. En cambio, si las dos terminan, vemos que ambas ejecutan la suma el mismo número de veces. □

Solución del Ejercicio 5.11. Supongamos que funG y funH calculen dos funciones f y g de $k+1$ argumentos, y que las MTs M_g y M_h actuen de la forma siguiente: M_g empieza su ejecución en la configuración $(q_g, \triangleright w \underline{\sqcup} n_1 \sqcup .. \sqcup n_k \sqcup y)$,

donde n_i es la representación del i-ésimo input de funF en la cinta, e y es la representación de y. Si $g(n_1, .., n_{k+1})$ está definida, M_g termina en $(q'_g, \triangleright w \underline{\ } g(n_1, .., n_{k+1}))$. Lo mismo vale para M_h. Esto quiere decir que las dos máquinas ignoran la parte de la cinta que está a la izquierda de la posición inicial de la cabeza, y la cabeza vuelve a esa misma posición inicial al acabar su computación. Los números están escritos con notación unaria.

M_f empieza su computación con la cinta en la configuración $(s, \triangleright \underline{\ } n_1, .., n_k)$ y con las siguientes operaciones:

- escribe el número 0 (que sería y) al final de la cinta;
- escribe otro número 0 (que sería x) al final de la cinta.

Empieza un bucle; al principio de cada iteración la cinta contiene $k+2$ números: los inputs $n_1, .., n_k$, el valor actual de y y el valor actual de x. En cada iteración:

- se copia los primeros $k + 1$ valores al final de la cinta;
- después de posicionar correctamente la cabeza, se ejecuta la computación de M_g; al final del trabajo de M_g, el número que aparece inmediatamente a la derecha de la cabeza es el resultado $g(n_1, .., n_k, y)$:
  - si es 1, entonces, se sale del bucle y se devuelve el valor del número $k + 2$ en la cinta (que sería x); esto se obtiene borrando todos los demás números y moviendo x hasta el principio de la cinta;
  - si no lo es, se borra este último número de la cinta e se continúa con la iteración del bucle;
- se vuelve a copiar los primeros $k + 1$ valores al final de la cinta;
- después de posicionar correctamente la cabeza, se ejecuta la computación de M_h; después de terminar la computación de M_h, los últimos dos números en la cinta son el actual valor de x y el resultado de $h(n_1, .., n_k, y)$;
- se suma estos dos números y el resultado se almacena en donde se guarda x;
- se incrementa y en uno; esto significa desplazar la representación de x una casilla a la derecha.

Hemos omitido muchos detalles pero el funcionamiento de M_f es bastante claro. La Máquina de Turing termina si y sólo si la rutina funF termina, y el resultado es el mismo para todo input. □

Solución del Ejercicio 5.13. La clave de este ejercicio es que una Máquina de Turing tiene un número finito de símbolos y estados, y cada uno de ellos puede representarse con un número en el programa WHILE. Asímismo, no hemos puesto ninguna limitación en el tamaño de los número que maneja el lenguaje WHILE, así que podemos pensar que en cada momento el contenido de la cinta se pueda modelizar con un número natural suficientemente grande.

Supongamos que la función de transición de M sea una tabla con cierto número de líneas, y que dos de ellas sean

estado actual	símbolo leído	acción	nuevo estado
q_5	σ_{14}	σ_{12}	q_{11}
q_2	σ_4	$\rightarrow$	q_9

El programa WHILE que simula M se obtiene con un bucle **while** en cuyo cuerpo se calculan los efectos de cada paso de M.

```
1  funF(x1,..,xk) {
2    estado := 0; // estado inicial
3    cinta := escribeInput(x1,..,xk);
4    posicion := 1; // primera posición después de ▷
5    simbolo := leeSimbolo(cinta,posicion);
6
7    while (¬ estadoFinal(estado)) do {
8      // hay un if-then-else para cada línea de la tabla de
           M
9      ...
10     // primera línea indicada arriba
11     if (simbolo=14 ∧ estado=5) then {
12       cinta := escribeSimbolo(cinta,posicion,12);
13       estado := 11;
14     }
15     ...
16     // segunda línea indicada arriba
17     if (simbolo=4 ∧ estado=2) then {
18       posicion := posicion+1;
19       estado := 9;
20     }
21   }
22
23   return leeOutput(cinta);
24 }
```

Necesitamos algunas rutinas que nos permitan leer y escribir en la cinta. Los números se representan en notación unaria con secuencias de símbolos σ_2 (cod-

ificado con el número 2); el símbolo $\sigma_0 = \text{␣}$ se representa con 0 mientras que ▷ se representa con 1. La variable sigma es el número de símbolos del alfabeto de M, y es conocida desde el principio.

```
escribeInput(x1,..,xk) {
  cinta := 1; // sólo ▷ en la primera casilla
  posicion := 2;
  for i=1 to x1+1 do {
    cinta := cinta + 2*sigma^posicion;
    posicion := posicion+1;
  }
  posicion := posicion+1; // espacio entre dos números
  for i=1 to x2+1 do {
    cinta := cinta + 2*sigma^posicion;
    posicion := posicion+1;
  }
  posicion := posicion+1; // espacio entre dos números
  ...
  for i=1 to xk+1 do {
    cinta := cinta + 2*sigma^posicion;
    posicion := posicion+1;
  }
  return cinta;
}
```

El resultado es un número probablemente muy grande cuya cifras, interpretadas en base sigma, representan cada una una casilla de la cinta, empezando por ▷.

```
leeSimbolo(cinta,posicion) {
  x := cinta;
  for i=1 to posicion do {
    x := x / sigma;
  }
  return x % sigma;
}
```

donde / es la división entera y % es el resto.

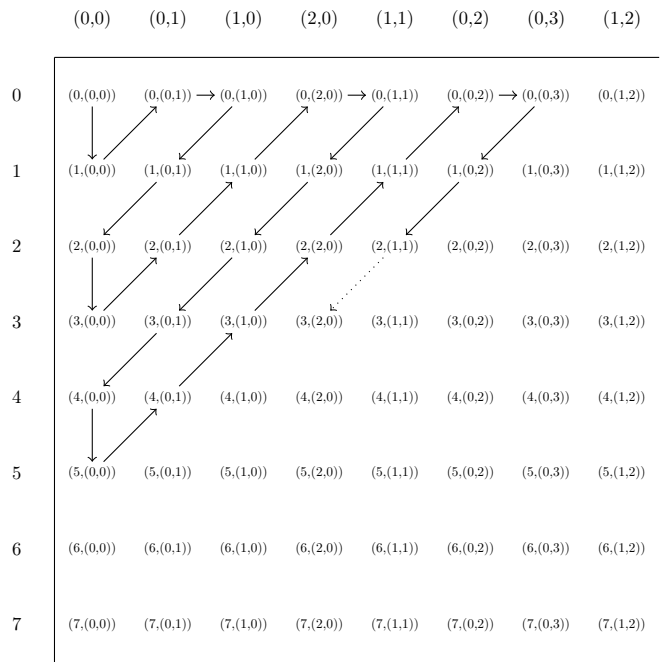
```
escribeSimbolo(cinta,posicion,simbolo) {
  s0 = leeSimbolo(cinta,posicion);
  return cinta+((simbolo-s0)*sigma^posicion);
}

leeOutput(cinta) {
```

```
7    x := -1;
8    y := cinta/sigma^2; // ignora las primeras dos casillas
9    while (y>0) do {
10     y := y/sigma;
11     x := x+1;
12   }
13   return x;
14 }
```

Finalmente, el predicado estadoFinal(estado) se limita a comprobar si estado es uno de os estados finales de M. No hay que olvidarse que, en toda esta transformación de M en un programa WHILE, se conoce *todo* de M: el número de estados y de símbolos que tiene, sus estados inicial y finales, y su función de transición. Lo que no se conoce es el input que, por esta misma razón, es también input de funF. El bucle principal de funF termina si y sólo si estado llega a representar un estado final de M, es decir, si M termina. □

Capítulo 7

Solución del Ejercicio 7.3. Dado que ya se sabe que $|\mathbb{N}| = \left|\mathbb{N}^2\right|$, para demostrar que $|\mathbb{N}| = \left|\mathbb{N}^3\right|$ lo que hay que hacer es construir una tabla similar donde un eje contiene la enumeración de $\mathbb{N}$ y el otro la enumeración de $\mathbb{N}^2$. Cada elemento del plano bidimensional corresponde unívocamente a una tripla de números naturales (por ejemplo, $(3,(0,1))$ corresponde a la tripla $(3,0,1)$), y siguiendo el mismo mecanismo de *dovetailing* se enumeran todos los elementos de $\mathbb{N}^3$.



La enumeración de $\mathbb{N}^3$ deseada viene a ser $(0,0,0)$, $(1,0,0)$, $(0,0,1)$, $(0,1,0)$, $(1,0,1)$, $(2,0,0)$, $(3,0,0)$, $(2,0,1)$, $(1,1,0)$, $(0,2,0)$, etc.

En el caso de $\mathbb{N}^4$ basta construir una tabla con los elementos de $\mathbb{N}^2$ en ambos ejes y repetir el mismo procedimiento. □

Solución del Ejercicio 7.5. La primera parte es fácil: siendo las funciones parciales un super-conjunto de las funciones totales, y siendo las totales no numerables, tampoco lo serán las parciales.

La segunda parte es menos evidente: a primera vista no puede haber tantas (más que enumerables) funciones con estas características porque hay un solo valor posible como output, así que sólo se pueden definir funciones constantes. Sin embargo, la variabilidad está en el dominio: para todo conjunto de número naturales $X \subseteq \mathbb{N}$, existe una función parcial f_X con dominio X y codominio $\{0\}$ tal que

$$\begin{array}{lcll} f_X(n) & = & 0 & \text{si } n \in X \\ f_X(n) & = & \text{indefinido} & \text{si } n \notin X \end{array}$$

Todas estas funciones son distintas y, al ser los X no enumerables (son lo elementos del conjunto potencia de $\mathbb{N}$), las f_X tampoco lo son. □

Solución del Ejercicio 7.7. La correspondencia biúnivoca es la función $f(x) = 1/x$, que es biunívoca porque ninguno de los dos conjuntos en cuestión contiene el 0. □

Solución del Ejercicio 7.9. Todas las funciones $f_m(n) = n + m$, para todo $m \in \mathbb{N}$, son calculables. Además, son estrictamente crecientes, y son infinitas porque hay una para cada m. □

Solución del Ejercicio 7.10. Este resultado se demuestra por diagonalización. Sea la tabla como siempre en este caso: la n-ésima línea contiene todos los valores de la n-ésima función monotona estrictamente creciente, llamada f_n. La función construida a partir de la diagonal de la tabla es f tal que $f(n) = \Sigma_{i=1}^{n} f_n(n) + n + 1$, es decir, el n-ésimo término de la secuencia de f es la suma de los primeros n elementos de la diagonal más $n + 1$. Con esta definición, f resulta ser monotona estrictamente creciente, y además no coincide con la diagonal en ningún punto. Por lo tanto, es una función distinta de cada una de las f_n.

Esto demuestra que las funciones monotonas estrictamente crecientes no son enumerables. Como consecuencia, sólo una parte infinitésima de las funciones monotonas estrictamente crecientes son calculables. □

Solución del Ejercicio 7.11. Una función parcial con dominio de un elemento se puede denominar como f_m^n, definida como

$$\begin{array}{lcll} f_m^n(x) & = & n & \text{si } x = m \\ f_m^n(x) & = & \text{indefinido} & \text{si } x \neq m \end{array}$$

Esto indica que hay una correspondencia biunívoca entre estas funciones parciales y $\mathbb{N}^2$: cada f_m^n corresponde al par (m, n). Por lo tanto, las funciones parciales con dominio de un elemento son enumerables. □

Solución del Ejercicio 7.14. Parecido al Ejercicio 7.11: en este caso cada función de $\{n_1, n_2\}$ a $\{0\}$ donde $n_1, n_2 \in \mathbb{N}$ corresponde al par de naturales (n_1, n_2), así que el conjunto de estas funciones resulta tener la misma cardinalidad que $\mathbb{N}^2$. □

Solución del Ejercicio 7.16. Lo que pide este ejercicio es una función f que, dado un conjunto de números, devuelva uno y un solo número natural, y cuya función inversa, dado un número, devuelva un solo conjunto de números. En otras palabras, f y su inversa f^{-1} establecen una correspondencia biunívoca entre números naturales y conjuntos finitos de números naturales. En general,

enumerar un conjunto A significa establecer una correspondencia biunívoca entre A y los números naturales.

La forma probablemente más directa de hacerlo es pensar que un conjunto finito $\{n_1, .., n_k\}$, donde n_k es el elemento máximo del conjunto, se puede representar como secuencia de $n_k + 1$ bits (es decir, representación binaria de un número) donde los bits en posición $n_1, .., n_k$ de la secuencia (empezando por la izquierda y considerando la primera posición como 0) son 1, y los demás son 0.

Por ejemplo, el conjunto $\{0, 1, 2\}$ se representa como la secuencia 111, mientras que $\{0, 3, 5, 9\}$ se representa como 1000101001. Entonces $f(A)$ devolverá, para todo conjunto A, este número escrito en decimal, es decir, $f(\{n_1, .., n_k\}) = \Sigma_{i=1}^{k} 2^{n_i}$. Por ejemplo, $f(\{0, 1, 2\}) = 2^0 + 2^1 + 2^2 = 1 + 2 + 4 = 7$ y $f(\{0, 3, 5, 9\}) = 2^0 + 2^3 + 2^5 + 2^9 = 1 + 8 + 32 + 512 = 553$.

La función inversa se limita a coger un número y ver cuáles bits de su representación binaria son 1. Ambas funciones son calculables. □

Capítulo 8

Solución del Ejercicio 8.5. La versión especializada más directa es

```
1 proc p1(n:int) returns int {
2   int m;
3   m := 7;
4   int j;
5   j := 1;
6   while (m<10) {
7     n := n+j;
8     j := j+1;
9     m := m+1
10  }
11  return n
12 }
```

que simplemente añade al principio de la rutina la declaración de una variable local `m` y su inicialización a 7. Esta versión no mejora en nada la eficiencia del algoritmo. De hecho, `p1` es incluso menos eficiente que `p`. En todo caso, generar `p1` a partir de `p` es una cosa que se puede hacer fácil y sistemáticamente de forma automática a través de un proceso de transformación de programas.

Usando transformaciones más refinadas (y, en muchos casos, de mucho más difícil automatización) se pueden obtener versiones especializadas más

eficientes. Por ejemplo:

```
1 proc p2(n:int) returns int {
2   int j;
3   j := 1;
4   for m=7 to 9 {
5     n := n+j;
6     j := j+1;
7   }
8   return n
9 }
```

transforma el bucle **while** en un **for** porque desde el principio de conoce el número de iteraciones. La rutina p2 en cierto sentido más sencilla porque el uso del bucle **for** garantiza la terminación sin necesidad de estudiar la semántica del programa, pero no gana mucho en eficiencia.

La tercera versión especializada es, a primera vista algo más difícil de obtener a través de un proceso automático: se necesita un análisis muy detallado de la rutina original para poder reemplazar m por j en el bucle.

```
1 proc p3(n:int) returns int {
2   for j=1 to 3 { n := n+j; }
3   return n
4 }
```

Finalmente, la última especialización es muy eficiente pero requiere entender perfectamente lo que p calcula. Como se verá en los próximos capítulos, esto es algo que no se puede garantizar en el caso general.

```
1 proc p4(n:int) returns int { return n+6 }
```

La versión p4 parece ser el programa más eficiente y más pequeño que implementa el mismo algoritmo que p cuando el valor de m es 7. Debido al Hecho 8.1, existen infinitas versiones especializadas de la rutina original. □

Solución del Ejercicio 8.8. Analizando la estructura lógica del enunciado, vemos que los hechos a demostrar son

- si A es vacío entonces es R.E.;
- si A es el codominio de una función calculable total entonces es R.E.;
- si A es R.E. entonces es vacío o el codominio de una función calculable total.

Lo primero que hay que hacer es retomar la definición de conjunto semidecidible o R.E.: A es recursivamente enumerable si existe una MT que lo semidecide.

El primer hecho es trivial: existe una Máquina de Turing que semidecide el conjunto vacío, y es cualquier máquina que no termina nunca.

Para el segundo hecho se nos pide demostrar, sabiendo que existe una función calculable total f cuyo codominio es A, que existe una MT que semidecide A. Que A es el codominio de f significa que para todo elemento $y \in A$ existe un x tal que $f(x) = y$ (por ejemplo, el codominio de la función $\lambda n.2n$ que multiplica por 2 son los números pares). Observamos que, al ser f calculable total, existe M_f que la calcula, y esta MT termina siempre. Así que la máquina M_A que semidecide A tiene que terminar si y sólo si su input y pertenece a A, y actúa de la siguiente manera:

- ejecuta un paso de la computación $M_f(0)$ (M_f aplicada al input 0);
- ejecuta un paso de $M_f(0)$ y un paso de $M_f(1)$;
- ejecuta un paso de $M_f(0)$, un paso de $M_f(1)$ y un paso de $M_f(2)$;
- ejecuta un paso de $M_f(0)$, un paso de $M_f(1)$, un paso de $M_f(2)$ y un paso de $M_f(3)$;
- ejecuta un paso de $M_f(0)$, un paso de $M_f(1)$, un paso de $M_f(2)$, un paso de $M_f(3)$ y un paso de $M_f(4)$;
- etc.

Es decir, M_A alterna la ejecución de M_f para todos los input con un método diagonal, obteniendo lo siguiente: toda computación $M_f(x)$ llega a dar m pasos tarde o temprano, para cualquier número m, a no ser que termine antes. Dado que todas estas computaciones terminan al ser f total, se trata de esperar hasta que esto suceda: si una computación $M_f(x)$ termina, y su resultado es y, entonces M_A termina su ejecución y el input y es aceptado; si el resultado no es y, se continúa hasta que otras computaciones terminen. Si y no pertenece a A, entonces tampoco pertenece al codominio de f, así que nunca se dará el caso que una computación $M_f(x)$ termine con resultado y. Por lo tanto, M_A está condenanda a la no terminación, pero esto es justamente lo que nos esperamos de una MT que semidecide un lenguaje.

La prueba del tercer hecho parte del hipótesis de recursiva enumerabilidad e intenta demostrar que $A = \emptyset$ o bien existe una función calculable total f con codominio A. Si A es R.E., entonces puede ser también recursivo: en

este caso (a) existe una M_A^R que lo decide. Si no es recursivo, (b) existe una M_A^{RE} que lo semidecide. En el caso (a), la MT M_f que calcula f se puede definir como la máquina que, dado un input n, de forma parecida a lo que acabamos de ver, ejecuta, de forma alternada, M_A^R sobre todos los posibles inputs, y espera que la *n-ésima* de estas computaciones termine en el estado de aceptación. Si la n-ésima computación que termina en el estado de aceptación es $M_A^R(m)$, entonces $f(n) = m$. Entonces, para todo input n, M_f ejecuta M_A^R sobre todos los inputs hasta que n computaciones hayan terminado en el estado de aceptación, y el resultado es distinto para todo n porque reproduce el orden en el que las computaciones de M_A^R terminan. El codominio de f viene a ser el conjunto de los inputs para los que M_A^R termina en el estado de aceptación, es decir, el mismo A. Notamos que todas las computaciones $M_A^R(m)$ terminan al ser A recursivo, así que f es calculable total, y tampoco hace falta alternar las computaciones de manera inteligente como acabamos de ver en la demostración del segundo hecho. En el caso (b), organizamos la demostración en dos pasos: (1) demostrar que existe una MT que *enumera* A; y (2) demostrar que existe una función calculable total f con codominio A. El paso (1) se obtiene definiendo la MT M_e como una máquina que ejecuta, de forma alternada, M_A^{RE} sobre todos los posibles inputs:

- ejecuta un paso de la computación $M_A^{RE}(0)$ (M_A^{RE} aplicada al input 0);
- ejecuta un paso de $M_A^{RE}(0)$ y un paso de $M_A^{RE}(1)$;
- ejecuta un paso de $M_A^{RE}(0)$, un paso de $M_A^{RE}(1)$ y un paso de $M_A^{RE}(2)$;
- ejecuta un paso de $M_A^{RE}(0)$, un paso de $M_A^{RE}(1)$, un paso de $M_A^{RE}(2)$ y un paso de $M_A^{RE}(3)$;
- ejecuta un paso de $M_A^{RE}(0)$, un paso de $M_A^{RE}(1)$, un paso de $M_A^{RE}(2)$, un paso de $M_A^{RE}(3)$ y un paso de $M_A^{RE}(4)$;
- etc.

y, cada vez que una de estas computaciones $M_A^{RE}(m)$ termina, escribe el input m (no el output) en la cinta. De esta forma, M_e enumera A. La MT M_f que calcula f se puede definir como la máquina que, dado un input n, espera a que M_e enumere el n-ésimo número, y devuelve ese número. Es total porque sabemos que todo conjunto RE que no es recursivo (caso (b)) es infinito, por lo que M_e tendrá siempre un número para devolver a M_f para todo n. □

Solución del Ejercicio 8.11. Si A y B son conjuntos RE, entonces existen dos MTs M_A y M_B que los semideciden. Es decir, para todo x, $M_A(x)$ termina si y sólo si $x \in A$ y $M_B(x)$ termina si y sólo si $x \in B$. Se puede definir una tercera Máquina de Turing M' con dos cintas tal que, dado un input x, ejecuta en cada una de las dos cintas y por separado (en paralelo) las computaciones de M_A y M_B sobre x. M' termina sólo cuando ambas computaciones terminan, es decir, espera a que tanto M_A como M_B hayan terminado su trabajo. Pero, por definición, esto sucede si y sólo si $x \in A$ y $x \in B$, es decir, si y sólo si $x \in A \cap B$. Por lo tanto, M' semidecide $A \cap B$, que se demuestra ser recursivamente enumerable. Demostrar que $A \cup B$ es también RE se consigue definiendo M'' parecida a M', pero con una diferencia: en lugar de esperar a que las dos computaciones terminen, M'' sólo espera a que una de las dos termine y, en este caso, aborta la otra computación y termina a su vez. La máquina M'' termina para x sí y sólo si una entre M_A y M_B termina para x, es decir, si $x \in A$ o $x \in B$. Por lo tanto, M'' semidecide el conjunto $A \cup B$, que también es recursivamente enumerable. □

Solución del Ejercicio 8.12. Lo que pide el enunciado es que se demuestre que para todo conjunto recursivo A, su complementario $\overline{A}$ es también recursivo. Que A sea recursivo implica que existe una MT M_A que lo decide: para todo input x, $M(x)$ termina en un estado de aceptación si $x \in A$, y en un estado de rechazo si $x \notin A$. Una máquina que decide $\overline{A}$ se otiene fácilmente a partir de M_A intercambiando estados de aceptación y de rechazo, por lo que M_B acepta todo input rechazado por M_A y rechaza todo input aceptado por M_A.

□

Solución del Ejercicio 8.15. Este ejercicio es un ejemplo de cómo ciertos problemas son más fáciles de lo que parecen. Aquí no se pide una función que genere el desarrollo decimal de un número real y diga algo sobre él. El punto es que el número real que nos interesa es uno: π, así que su desarrollo decimal viene a ser el mismo siempre.

Si para todo x aparecen x sietes consecutivos, entonces f es la función 1 constante, que es calculable. En caso contrario, existe un k tal que

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

y la función es otra vez calculable. Así que hemos contestado una pregunta sobre la calculabilidad de f sin necesidad de saber cómo es la función. □

Solución del Ejercicio 8.16. Si F fuera calculable, entonces también lo

sería

$$G(x) = \begin{cases} 1 & \text{si } F(x) = 0 \\ \text{indefinida} & \text{en los demás casos} \end{cases}$$

Al ser G calculable, tendrá que ser calculada por una MT; por lo tanto, G tendrá que ser alguna de las φ_m. Pero en este caso

$$\varphi_m(m) \text{ converge} \Leftrightarrow \varphi_m(m) = 1 \Leftrightarrow F(m) = 0 \Leftrightarrow \varphi_m(m) \text{ no converge}$$

que es absurdo. Por lo tanto, F no puede ser calculable. □

Capítulo 9

Solución del Ejercicio 9.3. Supongamos que P sea recursivo. Sea x el índice de la función calculable parcial

$$\varphi_x(y, z) = \begin{cases} z & \text{si } y \in K \\ \text{indefinida} & \text{en caso contrario} \end{cases}$$

Esta función es calculable parcial porque se puede semidecidir la pertenencia de un índice a K ejecutando la MT M_y sobre el mismo y. Por el Teorema del Parámetro, existe una $h(y)$ calculable total y biunívoca tal que $\varphi_{h(y)}(z) = \varphi_x(y, z)$ para todo z: esta $h(y)$ no es otra cosa que la $s(x, y)$ mencionada en el teorema, donde se ha fijado el valor de x. Así que tenemos

$$\varphi_{h(y)}(z) = \begin{cases} z & \text{si } y \in K \\ \text{indefinida} & \text{en caso contrario} \end{cases}$$

Ahora, $P(h(y)) = 1$ si y sólo si $\varphi_{h(y)}$ tiene codominio infinito, pero esto pasa si y sólo si $y \in K$ (si no, no estaría definida para ningún input, mientras que, si se elige el primer caso de la definición, se cubre todos los valores). Al ser K no recursivo, el predicado P tampoco puede serlo. El truco en esta demostración es usar el Teorema del Parámetro para que y deje de ser un parámetro de la función calculable cuyo índice $h(y)$ se pasa a P como input. De este modo, la pertenencia de y a K hace que, para todo z, se seleccione el primer caso de la definición de $\varphi_x(y, z)$, así que ésta nunca resulta ser indefinida y su codominio es infinito al generar todos los valores como output (codominio infinito). □

Solución del Ejercicio 9.4. Sea h la función calculable tal que

$$\varphi_{h(y)}(z) = \begin{cases} 1 & \text{si } y \in K \\ \text{indefinida} & \text{en caso contrario} \end{cases}$$

La calculabilidad de h la tenemos por el Teorema de Parámetro y un desarrollo parecido al de la solución del Ejercicio 9.3. Ahora, $P(h(y)) = 1$ si y sólo si $\varphi_{h(y)}$ es total, pero esto pasa si y sólo si $y \in K$. Al ser este último conjunto no recursivo, el primero (es decir, el predicado P) tampoco lo es. □

Solución del Ejercicio 9.5. Sea h la función calculable tal que

$$\varphi_{h(y)}(z) = \begin{cases} 0 & \text{si } y \in K \\ \text{indefinida} & \text{en caso contrario} \end{cases}$$

La calculabilidad de h la tenemos por el Teorema de Parámetro y un desarrollo parecido al de la solución del Ejercicio 9.3. Ahora, $P(h(y), 0, 0) = 1$ si y sólo si $\varphi_{h(y)}$ es la función 0-constante, pero esto pasa si y sólo si $y \in K$. Al ser K no recursivo, P tampoco puede serlo. □

Solución del Ejercicio 9.6. Sea h la función calculable tal que

$$\varphi_{h(y)}(z) = \begin{cases} 1 & \text{si } \varphi_y(y) \text{ converge} \\ \text{indefinida} & \text{en caso contrario} \end{cases}$$

La calculabilidad de h la tenemos por el Teoremal de Parámetro y un desarrollo parecido al de la solución del Ejercicio 9.3. Ahora, $P(h(y)) = 1$ si y sólo si $\varphi_{h(y)}$ es inyectiva, pero esto pasa si y sólo si φ_y diverge para el input y, es decir, si y sólo si $y \notin K$. Al ser $\overline{K}$ no recursivo, el predicado P tampoco lo es. □

Solución del Ejercicio 9.7. Que A sea recursivamente enumerable significa dos cosas (equivalentes): (1) que existe una MT que lo semidecide; y (2) que existe una MT que enumera (posiblemente en un tiempo infinito) todos los elementos de A. Así que de la existencia de una MT M_A que hace una de las dos cosas con A, tenemos que demostrar la existencia de otra MT M_B que hace una de las dos cosas con B.

La primera solución que se plantea es: enumerar los elementos de A (alternativa (2)), y, para cada elemento n, enumerar el dominio W_n de φ_n. Este enfoque tiene tres problemas.

- El primero es de fácil solución: al enumerar todos los W_n no tendríamos que repetir, porque B es definido como la unión. Este problema se soluciona considerando que cuando se genera un elemento m de W_n, el número de elementos generado hasta el momento en cualquiera de los W es finito, así que se puede comparar el nuevo elemento con todos los demás y descartarlo si es igual a uno que ya ha sido generado antes.

- El segundo problema es que hay que encontrar la manera de enumerar W_n. Esto se obtiene intercalando las computaciones de φ_n para cada uno de los inputs, como vimos en otras ocasiones:
  - un paso de $\varphi_n(0)$;
  - un paso de $\varphi_n(0)$ y un paso de $\varphi_n(1)$;
  - un paso de $\varphi_n(0)$, un paso de $\varphi_n(1)$ y un paso de $\varphi_n(2)$;
  - un paso de $\varphi_n(0)$, uno de $\varphi_n(1)$, uno de $\varphi_n(2)$ y uno de $\varphi_n(3)$;
  - etc.

  De esta forma se generan todos los m para los que $\varphi_n(m)$ converge, es decir, el conjunto W_n.

- El tercer problema, de más difícil solución, es que para cada n la enumeración de W_n tardaría, en general, un tiempo infinito, por lo que no se puede esperar a que termine la enumeración de W_n para empezar con la de $W_{n'}$ para otro índice n'. La solución es, otra vez, intercalar las computaciones. Sea C_n la enumeración de W_n, definida arriba ya teniendo en cuenta la solución del segundo problema. La enumeración de B se obtiene con:
  - un paso de C_0;
  - un paso de C_0 y un paso de C_1;
  - un paso de C_0, un paso de C_1 y un paso de C_2;
  - un paso de C_0, un paso de C_1, un paso de C_2 y un paso de C_3;
  - etc.

  De esta forma se permite generar tarde o temprano todos los elementos de B. Todo este proceso es calculable y la Máquina de Turing que hace esto demuestra que B es recursivamente enumerable.

□

Solución del Ejercicio 9.8. El conjunto complementario de A, es decir, el conjunto $\overline{A}$ de los índices de funciones no inyectivas, es recursivamente enumerable porque existe una forma efectiva de enumerarlo. Sea C_x la computación que intercala las ejecuciones de cada una de las $\varphi_x(n)$; cada vez que φ_x converge para algún input, se compara el resultado con los obtenidos anteriormente y, si es igual a alguno de ellos, se devuelve x como elemento de $\overline{A}$ al ser φ_x no

inyectiva. Para realizar todo el proceso hay que intercalar también todas las C_x del misma modo.

Si A fuera también recursivamente enumerable, entonces ambos conjuntos serían recursivos (Sección 8.3). Pero sabemos por el Ejercicio 9.6 que A no es recursivo; por lo tanto, tampoco puede ser recursivamente enumerable por lo que acabamos de decir. □

Solución del Ejercicio 9.9. Si TOT fuera R.E., entonces sería o vacío (y claramente no lo es) o el codominio de una f calculable total. Sea $g(x) = \varphi_{f(x)}(x) + 1$; esta función es calculable total porque los índices que produce f son índices de funciones calculables totales. Por lo tanto, g será una de las φ, concretamente $\varphi_{f(y)}$ para cierto y. Pero entonces $\varphi_{f(y)}(y) = g(y) = \varphi_{f(y)}(y) + 1$ que es absurdo, así que TOT no es R.E.

Si $\overline{TOT}$ fuera R.E., existiría m tal que $\overline{TOT}$ es el dominio W_m de φ_m. Entonces sea

$$\varphi_{h(x)}(y) = \begin{cases} 1 & \text{si } \varphi_m(x) \text{ converge} \\ \text{indefinida} & \text{en caso contrario} \end{cases}$$

Ahora $h(x) \in W_m$ si y sólo si $\varphi_{h(x)}(y)$ no es total, pero esto sucede si y sólo si $x \notin W_m$, y se deduciría que TOT es el dominio de la función calculable parcial $\varphi_m(h(\_))$ porque

$$x \in \mathsf{dom}(\varphi_m(h(\_))) \Leftrightarrow h(x) \in W_m \Leftrightarrow x \notin W_m \Leftrightarrow x \in TOT$$

Esto no puede ser porque acabamos de demostrar que TOT no es R.E., por lo que $\overline{TOT}$ tampoco lo es. □

Solución del Ejercicio 9.10. Si A fuera recursivamente enumerable, entonces existiría m tal que $A = W_m$ donde W_m es el dominio de φ_m. Esto sucede porque la Máquina de Turing que calcula φ_m y enumera A converge exactamente en los elementos de A, por lo que el dominio de φ_m es A. Sin embargo, sea

$$\varphi_{h(x)}(y) = \begin{cases} 1 & \text{si, para todo } z \text{ entre 0 e } y,\ \varphi_x(z) \text{ converge} \\ \text{indefinida} & \text{en caso contrario} \end{cases}$$

Esta función es calculable, y también lo es h (Solución del Ejercicio 9.3). Se da que $h(x) \in W_m$ si y sólo si el dominio $W_{h(x)}$ de $\varphi_{h(x)}$ es infinito, y esto sucede si y sólo si φ_x es una función total (porque si no lo es, existiría un z donde no converge, y $\varphi_{h(x)}(y)$ sería indefinida para todo $y > z$). En este caso se

deduciría que el dominio de la función calculable parcial $\varphi_m(h(\_))$ es $TOT = \{\ x \mid \varphi_x \text{ es total }\}$. Pero esto no puede ser al no ser TOT recursivamente enumerable (Ejercicio 9.9).

Por otro lado, si $\overline{A}$ fuera recursivamente enumerable, entonces existiría m tal que $A = W_m$. Sea, otra vez,

$$\varphi_{h(x)}(y) = \begin{cases} 1 & \text{si, para todo } z \text{ entre 0 e } y,\ \varphi_x(z) \text{ converge} \\ \text{indefinida} & \text{en caso contrario} \end{cases}$$

Se da que $h(x) \in W_m$ si y sólo si el dominio $W_{h(x)}$ de $\varphi_{h(x)}$ es finito, y esto sucede si y sólo si φ_x no es una función total (porque si es total tiene dominio infinito). En este caso se deduciría que el dominio de la función calculable parcial $\varphi_m(h(\_))$ es $\overline{TOT} = \{\ x \mid \varphi_x \text{ no es total }\}$. Pero esto no puede ser al no ser $\overline{TOT}$ recursivamente enumerable (Ejercicio 9.9). □

Capítulo 10

Solución del Ejercicio 10.1. Nunca se puede dar una división con denominador 0 porque la línea 2 asegura que `i` es par (también puede ser par negativo como -2, -4, etc.) al entrar en el bucle por primera vez, y la línea 6 garantiza que `i` sigue siendo par en cada iteración. Por lo tanto, `i-1` es impar, así que no puede ser 0 (aunque podría ser negativo). □

Solución del Ejercicio 10.4. Nunca se lanza una excepción de este tipo porque `x.f` se hace apuntar a un objeto en el heap antes de que sea leído. Notamos que el bucle no termina: cada vez que se crea un nuevo objeto la variable `x` apunta a este objeto (línea 5), por lo que la condición del bucle nunca se hace falsa. □

Solución del Ejercicio 10.6. Sí es posible, y de hecho sucede sin que se pueda evitar porque el valor de `i` al entrar en el bucle por primera vez es 0; esto hace que la condición `i>0` sea falsa y que se ejecute `j++` sin que `j` tenga ningún valor. □

Solución del Ejercicio 10.9. La línea 6 nunca se ejecuta porque

- `i` tiene un valor no negativo después de la primera línea;
- `x` recibe el máximo valor entre `i` (modificado) y `j`;

- si `j` es mayor que `i`, entonces `x` vale 10;
- si no, `x` vale lo mismo que `i`;
- en el primer caso, `x−i` vale entre 0 y 10;
- en el segundo caso, `x−i` vale exactamente 0;
- por lo tanto, la condición del bucle nunca se cumple.

La línea 6 (y todo el bucle) es código muerto. □

Capítulo 11

Solución del Ejercicio 11.7. Los números que aparecen en la expresión son 3, −2 y 5. Calcular la mejor aproximación posible de un número significa escoger un valor aproximado entre [**¿?**], [**n-0**], [**n-neg**], [**n-pos**], [**pos**], [**neg**], y [**0**] tal que (1) la aproximación representa el número (por ejemplo, [**pos**] representa 3 pero no representa 0); y (2) no hay otro valor que también lo representa pero es más preciso (por ejemplo, [**n-neg**] representa 5 pero [**pos**] es una aproximación mejor porque también lo representa y es más precisa al no representar el 0). Por lo tanto, la mejor aproximación de 3 es [**pos**], la de −2 es [**neg**] y la de 5 es [**pos**]. La expresión aproximada viene a ser [**pos**][+]([**neg**][∗][**pos**]), y para evaluarla hay que actuar según las usuales reglas de precedencia: primero calcular [**neg**][∗][**pos**] y luego aplicar [**pos**][+] al resultado. Multiplicar un número negativo por uno positivo siempre da un número negativo (y al no poder ser 0 ninguno de los dos, el producto tampoco puede serlo), así que [**neg**][∗][**pos**] es [**neg**]; sumar un número positivo a uno negativo puede dar cualquier cosa, así que [**pos**][+][**neg**] es [**¿?**], que es el resultado final. □

Solución del Ejercicio 11.8. Parecido al anterior. Aquí hay que introducir una versión aproximada del elevamiento a potencia, pero no es otra cosa que una generalización del producto. La mejor aproximación de 7 es [**pos**], y la de −6 es [**neg**]. Por lo tanto, la expresión aproximada a evaluar es $[\mathbf{pos}]^{[2]}[+][\mathbf{neg}]^{[2]}$, donde $^{[2]}$ es la versión aproximada del elevamiento al cuadrado. Ahora, un número positivo al cuadrado es positivo siempre, y un número negativo al cuadrado también es positivo siempre. Por lo tanto,

$$[\mathbf{pos}]^{[2]}[+][\mathbf{neg}]^{[2]} = [\mathbf{pos}][+][\mathbf{pos}] = [\mathbf{pos}]$$

dado que la suma de dos números positivos es positiva. □

Solución del Ejercicio 11.11. En este caso se dan los datos ya aproximados, pero el resto del procedimiento es el mismo. La sub-expresión $[\mathbf{n\text{-}0}]^{[2]}$ es positiva siempre porque un número positivo o negativo pero $\neq 0$ es estrictamente positivo si se eleva al cuadrado. Si a esto se suma un [**pos**], el resultado sigue siendo positivo; finalmente, sumarle un no-negativo [**n-neg**] da como resultado final el valor aproximado [**pos**]. □

Solución del Ejercicio 11.12. Parecido al anterior. La expresión entre paréntesis [**n-neg**][−][**n-pos**] da como resultado [**n-neg**] porque restar a $n \geq 0$ un número negativo o nulo resulta en un número no-negativo (no se puede decir que es estrictamente positivo porque $0 - 0 = 0$). Además, $[\mathbf{¿?}]^{[2]} = [\mathbf{n\text{-}neg}]$ porque el cuadrado de un número no puede ser negativo, y sumando los dos valores se obtiene [**n-neg**] como resultado final. □

Solución del Ejercicio 11.17. Se van a dar sólo algunos casos de la implicación.

[**pos**]	[→]	[**pos**]	=	[**pos**]
[**pos**]	[→]	[**0**]	=	[**0**]
[**pos**]	[→]	[**n-neg**]	=	[**n-neg**]
[**0**]	[→]	[**pos**]	=	[**pos**]
[**0**]	[→]	[**0**]	=	[**pos**]
[**0**]	[→]	[**n-neg**]	=	[**pos**]
[**n-neg**]	[→]	[**pos**]	=	[**pos**]
[**n-neg**]	[→]	[**0**]	=	[**n-neg**]
[**n-neg**]	[→]	[**n-neg**]	=	[**n-neg**]

□

Capítulo 12

Solución del Ejercicio 12.2. Una posible transformación daría

```
1  proc p(i:int) returns int {
2    var j:int, k:int;
3    j := 1;
4    if (i>0) then {
5      k := i;
6    } else {
7      if (i−10≤−5) then {
```

```
8           k := j−i;
9         } else {
10          k := j+i;
11        }
12      }
13      return k;
14    }
```

donde el comando C es k := j−i y C' es k := j+i. Sabiendo que i es no-positivo es la rama "else" de la sentencia condicional principal, el valor final aproximado de k es positivo en el caso de C y desconocido en el caso de C'. La condición del bucle introducido por la transformación no puede ser interpretada por $[\![\ ]\!]^{\#'}$ a pesar de la optimización. □

Solución del Ejercicio 12.5. $t(P)$ sería simplemente

```
1   proc p(x:int) returns int {
2     if (x≥0) then {
3       return 2;
4     } else {
5       return −x;
6     }
7   }
```

porque $[\![\ ]\!]^{\#}$ tiene que considerar ambas ramas, y en la rama "else" (que nunca se ejecuta porque x es positivo por hipótesis) el valor de retorno sería negativo. $t(P)$ no es ofuscado para $[\![\ ]\!]^{\#'}$ porque la información adicional que se puede usar en la rama "else" es que x es negativa, por lo que −x viene a ser positivo. Por lo tanto, la unión de las dos ramas da una valor de retorno positivo.

Una posibilidad para $t'(P)$ es

```
1   proc p(x:int) returns int {
2     if (x∗2≥0) then {
3       return 2;
4     } else {
5       return −x;
6     }
7   }
```

que "engaña" $[\![\ ]\!]^{\#'}$ porque la condición de la sentencia condicional no es inteligible, pero no $[\![\ ]\!]^{\#''}$ porque esta última es capaz de extraer más información de la condición booleana.

Al no haber un bucle en esta rutina, no se puede definir una transformación que ofusque para $[\![\ ]\!]^{\#''}$ pero no para $[\![\ ]\!]^{\#''}_{w}$ (a no ser que se introduzcan predicados opacos en los bucles, como pide el Ejercicio 12.8). Por lo tanto, sólo se puede encontrar una tercera transformación que ofusque la rutina para ambas. La existencia de esta transformación depende de cuánto se haya conseguido optimizar $[\![\ ]\!]^{\#''}$ con respecto a $[\![\ ]\!]^{\#'}$, es decir, en qué casos $[\![\ ]\!]^{\#''}$ puede extraer información valiosa de las condiciones booleanas. Por ejemplo,

```
1  proc p(x:int) returns int {
2    if (x+1−1≥0) then {
3      return 2;
4    } else {
5      return −x;
6    }
7  }
```

es una ofuscación efectiva si la semántica aproximada no simplifica las expresiones aritméticas constantes. Si lo hace, entonces $+1-1$ vendría a ser $+0$, por lo que la condición sería verdadera siempre, también desde el punto de vista del analizador. □