Diagonalización y reducciones

Lógica y Computabilidad

Julián Dabbah (prestada de Franco Frizzo y Herman Schinca)

8 de septiembre de 2017

Ejercicio 1: Diagonalización

La **diagonalización** es un esquema de demostración que resulta útil para probar que no son computables [1] algunas funciones, en especial, aquellas cuyo **significado** involucra, de alguna manera, el resultado de algún programa sobre alguna entrada. Hacemos énfasis en la palabra *significado* porque, en principio, entendemos las funciones con su interpretación matemática más general, es decir, como elementos que describen relaciones o correspondencias entre elementos de dos conjuntos sin importarnos cómo esté expresada esta relación. Es decir, no nos importa cómo esté definida la función mientras podamos entender qué elemento se relaciona con cuál. Más sobre esto al final de la clase.

Volviendo a la forma en la que utilizaremos la diagonalización para escribir demostraciones, las demostraciones que obtendermos serán por el absurdo y este absurdo provendrá de haber supuesto que la función en cuestión era computable y quedará en evidencia cuando la evaluemos en el número de algún programa que la computa. Más precisamente:

- Tendremos f (cuyo significado depende de alguna manera del comportamiento que tendría un programa sobre alguna entrada) que querremos probar no computable.
- Supondremos f computable.
- Construiremos f' que deberá ser computable si f es computable.
  - Podemos definir f' a partir de f y otras funciones computables de manera computable (usando composición, recursión, minimización, etc.), o directamente, escribiendo un programa que use f.
  - La idea es definir una f', usando f, que dependa del resultado de un programa y que su valor sea "contradictorio" con el resultado del programa que toma como parámetro. **(!)**
- Como f' será computable, pues estaremos suponiendo que f lo es, existirá un programa que la compute, y este programa tendrá un número: e.
- Observaremos $f'(e)$ y obtendremos una contradicción. Como el significado de f' dependía del resultado de un programa, al mirar f' en e, el valor de f', que coincide con el resultado de evaluar el programa e en e, dependerá del comportamiento del e-ésimo programa evaluado en e. En definitiva, el programa que *calcula* el valor de f' coincide con el programa que *determina* el valor de f', y habíamos elegido f' para que estos dos fueran contradictorios.

[1] Más en general: para probar que una función no pertenece a una clase dada

Ejercicio 1.1

Demostrar que la siguiente función no es computable.

$$f_1(x) = \begin{cases} 1 & \text{si } \Phi_x^{(1)}(x) \downarrow \\ 0 & \text{en otro caso} \end{cases}$$

Resolución

Suponemos que f_1 es computable (sabemos que es total por su definición), y llegamos a un absurdo. Lo haremos construyendo un programa "rebelde": uno tal que, evaluado en *su propio número*, haga lo contrario a lo que dice f_1.

El primer indicio es ver qué observa f_1 respecto del comportamiento del programa: vemos que el valor de f_1 depende de si el programa x terminará o no al evaluarse en x.

La intuición nos dice, entonces, que tenemos que conseguir una función f_1' que cuando el programa que recibe termina, ella se indefina, y viceversa. Veamos cómo podemos construir esta tal función.

Si f_1 es computable, podemos definir el siguiente programa P_1:

$$\begin{array}{ll} & Z_1 \leftarrow f_1(X_1) \\ [A] & \text{IF } Z_1 \neq 0 \text{ GOTO } A \end{array}$$

Por inspección del programa P_1, podemos ver que para todo x se cumple que

$$\Psi_{P_1}^{(1)}(x) \downarrow \quad \Leftrightarrow \quad f_1(x) = 0$$

O lo que es lo mismo, tenemos definida una función parcial computable

$$f_1'(x) = \Psi_{P_1}^{(1)}(x) = \begin{cases} 0 & \text{si } f_1(x) = 0 \\ \uparrow & \text{en otro caso} \end{cases}$$

Además, por definición de f_1, para todo x vale

$$f_1(x) = 0 \quad \Leftrightarrow \quad \Phi_x^{(1)}(x) \uparrow$$

Por lo tanto, para todo x tenemos

$$\Psi_{P_1}^{(1)}(x) \downarrow \quad \Leftrightarrow \quad \Phi_x^{(1)}(x) \uparrow$$

En particular, para $e = \#(P_1)$ nos queda

$$\Phi_e^{(1)}(e) \downarrow \quad \Leftrightarrow \quad \Phi_e^{(1)}(e) \uparrow$$

lo cual es un absurdo.

Ejercicio 1.2

Demostrar que la siguiente función no es computable.

$$f_2(x, y) = \begin{cases} 0 & \text{si } \Phi_x^{(1)}(y) \text{ es par} \\ 1 & \text{en otro caso} \end{cases}$$

Resolución

Suponemos que f_2 es computable.

Al igual que antes, observemos que el valor de f_2 depende de si el programa x producirá un resultado par si lo evaluáramos sobre y. Con la misma idea del ejercicio anterior, ahora queremos una función parcial computable que si el programa que observa produce un resultado par, ella produzca un resultado impar.

Definimos, a partir de f_2, el siguiente programa P_2:

$$\begin{aligned} & \mathrm{Z}_1 \leftarrow f_2(X_1, X_1) \\ & \text{IF } \mathrm{Z}_1 \neq 0 \text{ GOTO P} \\ & \mathrm{Y} \leftarrow 13579 \\ & \text{GOTO E} \\ [P]\ & \mathrm{Y} \leftarrow 2468 \end{aligned}$$

En definitiva, la función computada por P_2 es

$$\Psi^{(1)}_{P_2}(x) \quad = \quad \begin{cases} 13579 & \text{si } f_2(x,x) = 0 \\ 2468 & \text{en otro caso} \end{cases}$$

Observemos que $\Psi^{(1)}_{P_2}(p)$ puede saber si evaluar el programa p en p producirá un resultado par gracias a que estamos suponiendo que f_2 es computable, y si esto pasa, toma un valor impar.

Sea $e = \#(P_2)$. Veamos qué sucede con $\Psi^{(1)}_{P_2}(e)$.

Por definición de P_2 tenemos que

$$\Psi^{(1)}_{P_2}(e) = 13579 \quad \Leftrightarrow \quad f_2(e,e) = 0$$

Por otro lado, por definición de f vale que

$$f_2(e,e) = 0 \quad \Leftrightarrow \quad \Psi^{(1)}_{P_2}(e) \text{ es par}$$

Siguiendo la cadena de equivalencias, se obtiene la siguiente contradicción:

$$\Psi^{(1)}_{P_2}(e) = 13579 \quad \Leftrightarrow \quad \Psi^{(1)}_{P_2}(e) \text{ es par}$$

la cual provino de suponer que f era computable.

Ejercicio 1.3

Demostrar que la siguiente función no es computable.

$$f_3(x) = \begin{cases} 3x & \text{si } \Phi^{(1)}_x(x) = x \\ 2x & \text{en otro caso} \end{cases}$$

Resolución

Suponemos que f_3 es computable.

Observamos que $f_3(x)$ varía según si el programa x, si lo evalúaramos en x, terminaría y devolvería x, o no.

Luego, intentemos construir un programa maldito que haga lo contrario a esto y, además, use f_3. Intuitivamente, querríamos definir un programa que cuando $f_3(x) = 3x$, es decir, cuando x si lo evaluáramos en x terminaría y daría x, no termine, y cuando $f_3(x) = 2x$ termine y devuelva x. Luego, construimos el siguiente programa P_3:

$$\begin{array}{ll} & Z_1 \leftarrow f_3(X_1) \\ [A] & \text{IF } Z_1 \neq 2 \times X_1 \text{ GOTO } A \\ & Y \leftarrow X_1 \end{array}$$

Inspeccionando el programa P_3, podemos ver que hace lo que queríamos, *siempre que* $X_1 \neq 0$. En definitiva, para todo x se cumple que

$$\Psi^{(1)}_{P_3}(x) \downarrow \quad \Leftrightarrow \quad f_3(x) = 2x \quad \text{ó} \quad x = 0$$

Por definición de f_3, para todo x vale

$$f_3(x) = 2x \quad \Leftrightarrow \quad \Phi^{(1)}_x(x) \uparrow \quad \text{ó} \quad \Phi^{(1)}_x(x) \neq x$$

Por lo tanto, para todo x

$$\Psi^{(1)}_{P_3}(x) \downarrow \quad \Leftrightarrow \quad \Phi^{(1)}_x(x) \uparrow \quad \text{ó} \quad \Phi^{(1)}_x(x) \neq x \quad \text{ó} \quad x = 0$$

Veamos ahora qué sucede cuando el programa recibe como entrada su propio número. Sea $e = \#(P_3)$

$$\Phi^{(1)}_e(e) \downarrow \quad \Leftrightarrow \quad \Phi^{(1)}_e(e) \uparrow \quad \text{ó} \quad \Phi^{(1)}_e(e) \neq e \quad \text{ó} \quad e = 0$$

Ahora bien,

- sabemos que $e \neq 0$, porque $e = \#(P_3)$ y P_3 no es el programa vacío, y
- sabemos que, en el caso de que $\Phi^{(1)}_e(e) \downarrow$, $\Phi^{(1)}_e(e) \neq e$ es falso, porque de terminar, P_3 devuelve siempre su entrada.

De esta manera, llegamos al absurdo

$$\Phi^{(1)}_e(e) \downarrow \quad \Leftrightarrow \quad \Phi^{(1)}_e(e) \uparrow$$

de donde concluimos que f_3 no es computable.

¿Qué otros programas podríamos haber construido? ¿Qué podríamos haber cambiado? ¿Cómo cambiaba el argumento?

Ejercicio 2: Reducciones

La **reducción** es una técnica que nos permite demostrar que cierta función no es computable a partir de otra función que ya sepamos que no lo es.

Para demostrar que una función g no es computable usando una reducción, elegimos una f que ya sepamos que no es computable y a partir de g construimos una función g' que se comporte como f, y

que además podamos demostrar que es computable suponiendo que g lo es. Luego, si suponemos que g es computable, tendremos que g' y por lo tanto f son computables, lo cual es una contradicción que habrá provenido de suponer g computable.

Intuitivamente, podemos pensar que g es *más general* que f en el sentido de que nos alcanza con usar g de una manera particular para resolver el mismo problema que resuelve f y que ya sabemos que no se puede computar.

Ejercicio 2.1

Demostrar que la siguiente función no es computable.

$$g_1(x,y) = \begin{cases} 1 & \text{si } \Phi_x^{(1)}(y)\downarrow \\ 0 & \text{en otro caso} \end{cases}$$

Resolución

Definimos $g_1'(x) = g_1(x,x) = g_1(u_1^1(x), u_1^1(x))$. Si suponemos g_1 computable, es claro que g_1' es computable pues se obtiene a partir de aquélla y composiciones con funciones iniciales. Como además también es claro que $g_1' = f_1$, tenemos que si g_1 es computable, entonces f_1 es computable, lo cual es una contradicción pues ya vimos que f_1 no lo es.

Ejercicio 2.2

Demostrar que la siguiente función no es computable.

$$g_2(x_1,y_1,x_2,y_2) = \begin{cases} 1 & \text{si } \Phi_{x_1}^{(1)}(y_1) \text{ es divisible por } \Phi_{x_2}^{(1)}(y_2) \\ 0 & \text{en otro caso} \end{cases}$$

Resolución

Intentaremos reducir la función f_2 a la función g_2 de manera computable. De esta forma, si g_2 fuera computable, f_2 también lo sería. Parece razonable pensar que, si podemos computar si el resultado de un programa es divisible por el resultado de otro, en particular podremos computar si es divisible por 2. O lo que es lo mismo, que si no podemos saber si un programa va a producir un resultado par, menos podríamos saber si el resultado de un programa será divisible por el resultado de otro.

Sea e_{id} el número de algún programa que compute la función identidad (existe, pues es primitiva recursiva y por lo tanto, computable).

Luego, podemos definir la siguiente función $h : \mathbb{N}^2 \to \mathbb{N}$ como

$$h_2(x,y) = \alpha\big(g_2(x,y,e_{id},2)\big)$$

Dado que suponemos g_2 computable y que α es una función primitiva recursiva, h_2 también debe ser computable. Por definición de h_2

$$h_2(x,y) = 0 \quad \Leftrightarrow \quad \alpha\big(g_2(x,y,e_{id},2)\big) = 0$$

Por lo tanto, como consecuencia de la definición de α, se tiene que

$$h_2(x,y) = 1 \quad \Leftrightarrow \quad g_2(x,y,e_{id},2) \neq 0$$

Además, por cómo fue definida g_2, $g_2(x_1, y_1, x_2, y_2) \neq 0$ si y solo si $g_2(x_1, y_1, x_2, y_2) = 1$. En consecuencia,

$$h_2(x, y) = 0 \quad \Leftrightarrow \quad g_2(x, y, e_{id}, 2) = 1$$

Siguiendo ahora con la definición de g_2, tenemos que

$$h_2(x, y) = 0 \quad \Leftrightarrow \quad \Phi_x^{(1)}(y) \text{ es divisible por } \Phi_{e_{id}}^{(1)}(2)$$

o, lo que es lo mismo,

$$h_2(x, y) = 0 \quad \Leftrightarrow \quad \Phi_x^{(1)}(y) \text{ es divisible por } 2$$

con lo cual nos sacamos de encima los dos últimos argumentos de g_2.

Definitivamente, esta reducción no es la única posible. ¿Qué otras podríamos haber hecho? ¿Qué otros números podríamos haber elegido para poner en los dos últimos parámetros de g_2?

De acuerdo a la definición de f_2 llegamos entonces a que

$$h_2(x, y) = 0 \quad \Leftrightarrow \quad f_2(x, y) = 0$$

(y queda como ejercicio alicar el mismo razonamiento para probar que $h_2(x, y) = 1 \quad \Leftrightarrow \quad f_2(x, y) = 1$). Con lo cual, finalmente, podemos deducir que $h_2 = f_2$, y por lo tanto, debe ser f_2 computable.

En definitiva, al suponer g_2 computable, encontramos una forma de computar f_2. Como sabíamos de antemano que esto no era posible, podemos concluir que g_2 no es computable.

Ejercicio 3

Dadas dos funciones parciales $f : \mathbb{N} \to \mathbb{N}$ y $g : \mathbb{N} \to \mathbb{N}$, decimos que g *domina fuertemente* a f si $g(x) \geq f(x)$ para todo x tal que $x \in \operatorname{Dom} f$ y $x \in \operatorname{Dom} g$.

Sea $h : \mathbb{N} \to \mathbb{N}$ definida como

$$h(x) = \begin{cases} \min\limits_t \operatorname{STP}^{(1)}(x, x, t) & \text{si } \operatorname{HALT}(x, x) \\ \uparrow & \text{en otro caso} \end{cases}$$

Demostrar que si $g : \mathbb{N} \to \mathbb{N}$ es una función total que domina fuertemente a h, entonces g no es computable.

Resolución

Supongamos que g es computable y veamos que entonces podemos computar el predicado $\operatorname{HALT}'(x) = \operatorname{HALT}(x, x)$.

Algunas observaciones, sobre el *significado* de h y de g:

- $h(x)$ es la (mínima) cantidad de pasos que necesita el programa x con entrada x para terminar. Si el programa x con entrada x no termina, esta cantidad no está definida y tampoco h. **(*)**

  $\Rightarrow$ Si pudiéramos saber si $h(x)$ está definida, podríamos saber si $\Phi_x(x)$ está definida, o sea, $\operatorname{HALT}'(x)$.

- Como g domina fuertemente a h:

  $\Rightarrow$ cuando h está definida, $g(x) \geq h(x)$ y, por lo tanto, $g(x)$ es una cantidad suficiente de pasos para que el programa x con entrada x termine. **(**)**

$\Rightarrow$ cuando h no está definida, g está definida, es decir, devuelve un valor, pero es fruta.

$\Longrightarrow$ Si pudiéramos diferenciar estos casos, podríamos saber si h está definida.

Veamos que, efectivamente, podemos distinguir estos casos:

- Si $h(x)\downarrow$, entonces $\mathrm{STP}^{(1)}(x, x, g(x))$, por **(**)**
- Si $h(x)\uparrow$, entonces $\neg\,\mathrm{STP}^{(1)}(x, x, t)$ para cualquier t, en particular, $\neg\,\mathrm{STP}^{(1)}(x, x, g(x))$, por **(*)**

$\Longrightarrow$ En definitiva, $h(x)\downarrow$ sii $\mathrm{STP}^{(1)}(x, x, g(x))$

Luego, podemos definir $h'(x) = \mathrm{STP}^{(1)}(x, x, g(x))$ y tenemos:

$$h'(x) = \begin{cases} 1 & \text{si } h(x)\downarrow \\ 0 & \text{en otro caso} \end{cases}$$

O, lo que es lo mismo, como vimos en la primera observación,

$$h'(x) = \begin{cases} 1 & \text{si } \mathrm{HALT}'(x) \\ 0 & \text{en otro caso} \end{cases}$$

con lo cual $h' = \mathrm{HALT}'$, y, además, habíamos definido $h'(x) = \mathrm{STP}^{(1)}(x, x, g(x))$ (es decir, a partir composición de funciones primitivas recursivas y computables). Luego, h' debe ser computable, lo que es un absurdo que provino de suponer g computable.

Ejercicio 4

Demostrar o refutar las siguientes afirmaciones.

(a) Si $f : \mathbb{N}^n \to \mathbb{N}$ es una función total tal que, para alguna constante $k \in \mathbb{N}$, $f(x_1, \ldots, x_n) \leq k$ para todo $(x_1, \ldots, x_n) \in \mathbb{N}^n$, entonces f es computable.

(b) La función

$$g(x) = \begin{cases} 1 & \text{si } \mathrm{HALT}(650, 323) \\ 0 & \text{en otro caso} \end{cases}$$

es computable.

(c) La función

$$D(x) = \begin{cases} 1 & \text{si Dios existe} \\ 0 & \text{en otro caso} \end{cases}$$

es computable.

(d) La función dada por $h(x) = \mathrm{HALT}(\mathrm{HALT}(x))$ es primitiva recursiva.

Resolución

(a) **Falso**.

Sea el siguiente contraejemplo

$$f(x) = \begin{cases} 1 & \text{si HALT}(x, x) \\ 0 & \text{en otro caso} \end{cases}$$

En este caso $f(x) \leq 1$ para todo x. Sin embargo f no es computable.

(b) **Verdadero**.

Al analizar la condición de g se observa que:

$$\text{HALT}(650, 323) \Leftrightarrow \text{El programa número } 323 \text{ con entrada } 650 \text{ se detiene.}$$

La sentencia que expresa $\text{HALT}(650, 323)$ es o bien verdadera o bien falsa para todo x. En decir, a pesar de que no sepamos cuál es el valor exacto de $\text{HALT}(650, 323)$, sabemos que es constante: 1 ó 0.

En el primer caso, g es computada por el programa

$$Y \leftarrow 1$$

En cambio, si es falsa, g es computada por el programa vacío.

En cualquier caso, g se trata de una función computable: la función constante 1 o la función constante 0.

(c) **Verdadero**.

La resolución es análoga al ítem anterior. O bien Dios existe o bien Dios no existe. Sea cual fuere La Verdad, la función D será la constante 1 o la constante 0. En cualquiera de ambos casos, D será computable. Dejamos como ejercicio determinar qué programa computa efectivamente D.

(d) **Verdadero**.

Observemos el *significado* de h. Con un argumento similar a los anteriores, si bien en principio no sabemos cuánto vale $\text{HALT}(x)$, éste no puede ser otra cosa que 0 o 1. Por lo tanto, para cualquier x, $h(x) = \text{HALT}(0)$, o bien $h(x) = \text{HALT}(1)$. Queda como ejercicio decodificar los programas con número 0 y 1, pero son lo suficientemente simples como para no contener ningún ciclo, y por lo tanto, terminar siempre. En definitiva, en cualquiera de los dos casos, $h(x) = 1$, con lo cual, h es una función constante, y por lo tanto, primitiva recursiva.