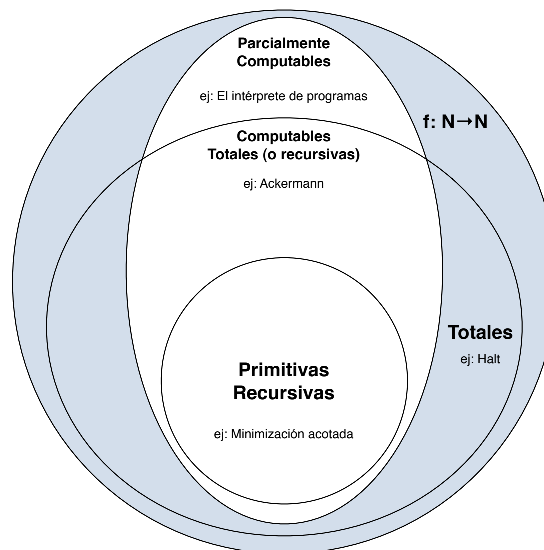
Clase práctica 5 (computabilidad)
Lógica y Computabilidad

Facundo Carreiro

1. Repaso de clases

Diagrama de Venn de clases: Ubiquemos en un diagrama de Venn las distintas clases de funciones de $\mathbb{N} \to \mathbb{N}$ que vimos hasta ahora[1]:



- **Primitivas Recursivas** (programas FOR)
  - Funciones iniciales
  - Composición
  - Recursión primitiva
- **Computables** (programas que terminan)
  - Funciones iniciales
  - Composición
  - Recursión primitiva
  - *Minimización garantizada*
- **Parcialmente Computables** (todos los programas posibles en el lenguaje $\mathcal{S}$)
  - Funciones iniciales
  - Composición
  - Recursión primitiva
  - *Minimización no acotada no garantizada*

[1] En color están las funciones **no** computables. Notar que las funciones totales y las computables no coinciden.

2. Reducir Halt

Ej 1 pag. 69 (Davis): Demostrar que $HALT(x, x)$ no es computable.

Resolución: Lo resolvemos por el absurdo, suponemos que $HALT(x, x)$ es computable, y planteamos la función:

$$f(x) = \begin{cases} \uparrow & \text{si } HALT(x, x) \\ 0 & \text{si } \neg HALT(x, x) \end{cases}$$

Si $HALT$ es computable esta función es parcialmente computable ya que usa $HALT$ (que suponemos computable), division por casos (p.r.), composición (p.r.), 0 (p.r.) y $\neg$ (p.r.).

Al ser parcialmente computable existe un programa $\mathcal{P}$ que la computa en el lenguaje $\mathcal{S}$. Si $\#(\mathcal{P}) = e$, analizando la detención de ese programa vemos que

$$HALT(x, e) \iff \neg HALT(x, x)$$

ya que el programa (por la definición de f) debería terminar sólo si se cumple $\neg HALT(x, x)$. En este caso fijamos la y pero la relación debería cumplirse para todo x, y por lo tanto podemos instanciar a $x = e$. En este caso queda

$$HALT(e, e) \iff \neg HALT(e, e)$$

Y llegamos a un absurdo que proviene de suponer a $HALT(x, x)$ como computable. $\square$

Ej 2 pag. 69 (Davis): Demostrar que

$$NHALT(x, y) = \begin{cases} 1 & \text{si } \neg HALT(x, y) \\ 0 & \text{si } HALT(x, y) \end{cases}$$

no es computable.

Resolución: Suponemos que si es computable, en tal caso es un predicado que es verdadero si y solo si el programa y **no** para con entrada x. Podemos definir

$$HALT(x, y) = \neg NHALT(x, y)$$

y de esta manera estaríamos definiendo $HALT$ a partir de una función computable ($NHALT$) y una p.r. ($\neg$). Podríamos concluir que $HALT(x, y)$ es computable pero eso es falso! (lo acabamos de ver) entonces llegamos a un absurdo que proviene de suponer $NHALT(x, y)$ computable. $\square$

Ej 4 pag. 70 (Davis): Decidir si $f(x_1, ..., x_n) \leq k$ con k constante y f total implica que f es computable o no.

Resolución: La implicación es falsa ya que como (contra)ejemplo tenemos a $HALT(x, y)$ que es una función total cuya imagen es $\{0, 1\}$. Podemos acotar los elementos de la imagen por $k = 1$ y sin embargo la función no es computable.

Ej 5 pag. 70 (Davis): Mostrar que

$$f(x) = \begin{cases} x & \text{si la conjetura de Goldbach es verdadera} \\ 0 & \text{si no} \end{cases}$$

es primitiva recursiva. La conjetura de Goldbach es "todo $n \geq 4$ *par* puede ser expresado como suma de dos primos" y aún no se sabe si es verdadera o falsa.

Resolución: Este ejercicio es para ejemplificar la "debilidad" de pedir que una función sea computable. Recordemos que una función será computable si **existe** un método efectivo (para nosotros todo lo equivalente al lenguaje $\mathcal{S}$) que arroje el resultado especificado por la función. No necesariamente debemos saber cuál si no probar su existencia.

En este caso, la conjetura será verdadera o bien falsa, eso no lo sabemos. Pero sabemos que si es verdadera el programa que compute la **función identidad** computará la función f y en el caso de que sea falsa el **programa nulo** la computará.

Verdadera	**Falsa**
$f(x) = \begin{cases} x & \text{si true} \\ 0 & \text{si false} \end{cases} \Rightarrow f(x) = x$	$f(x) = \begin{cases} x & \text{si false} \\ 0 & \text{si true} \end{cases} \Rightarrow f(x) = 0$

No sabemos en cuál de los dos casos estamos pero si sabemos que no importa el caso podemos computar la función.

Se prueba que es p.r. porque tanto la identidad como la función nula son p.r. □

3. Universalidad

Ej 2a pag. 77 (Davis): Decidir si

$$f(x) = \begin{cases} 1 & \text{si } \Phi_x(x) \downarrow \\ \uparrow & \text{si no} \end{cases}$$

es parcialmente computable, computable, p.r. o no computable.

Resolución: Primero vemos que f está definida de manera parcial ya que uno de los casos se indefine. Esto ya descarta la posibilidad de que la función sea computable total, en el mejor caso será parcialmente computable[2]. También vemos que se encuentra la función $\Phi_y(x)$ — que es una notación alternativa para $\Phi^{(1)}(x, y)$ —, nosotros sabemos que esa función es parcialmente computable. La función pareciera ser parcialmente computable ya que usa funciones parcialmente computables, composición y división por casos, para convencernos podemos escribir un programa en $\mathcal{S}$ que lo compute

```
Y ← 1
Z ← Φ_x(x)
```

Justificación: Analizando el programa vemos que primero ponemos 1 en la variable de salida y luego llamamos al intérprete. Por lo tanto si el intérprete termina devolverá 1 y si no termina el programa se cuelga. El comportamiento coincide con la definición de $f(x)$.

Ej 2b pag. 77 (Davis): Sea $A = \{a_1, ..., a_n\}$ finito tal que $\Phi(a_i, a_i) \uparrow$ ver que

$$f(x) = \begin{cases} 1 & \text{si } \Phi_x(x) \downarrow \\ 0 & \text{si } x \in A \\ \uparrow & \text{si no} \end{cases}$$

es parcialmente computable.

[2]Esto es cierto siempre y cuando sea posible hacer verdadera la guarda del caso indefinido, si la guarda del caso indefinido nunca puede ser verdadera entonces el caso debería ser descartado.

Resolución: Esta función parece una pequeña variación de la anterior. ¿Cómo podemos modificar el programa para cumplir con la nueva definición? El problema que tenemos es el de decidir si un elemento pertenece al conjunto o no. Uno puede pensar que hay que usar la propiedad que cumplen los elementos del conjunto: $\Phi(a_i, a_i)\uparrow$. En tal caso deberíamos tener que probar si el programa no termina y en tal caso devolver 0. ¿Está bien?

No. Primero que nada, en general **no podemos** decidir si un programa va a terminar o no, **nunca** podemos escribir algo como

```
IF Φ_y(x) ↑ GOTO NOTERMINA
```

ya que, si el intérprete no termina todo el programa se cuelga. Segundo, incluso si pudieramos hacer eso no alcanza ya que el programa podría no terminar y no pertenecer al conjunto.

En este caso debemos notar que los elementos del conjunto están *fijos* y son finitos elementos entonces podemos reescribir la función como

$$f(x) = \begin{cases} 1 & \text{si } \Phi_x(x)\downarrow \\ 0 & \text{si } x = a_1 \\ 0 & \text{si } x = a_2 \\ & \vdots \\ 0 & \text{si } x = a_n \\ \uparrow & \text{si no} \end{cases}$$

por lo tanto podemos escribir un nuevo programa en $\mathcal{S}$ de la siguiente manera

```
IF X = a_1 GOTO FIN
IF X = a_2 GOTO FIN
⋮
IF X = a_n GOTO FIN
Y ← 1
Z ← Φ_x(x)
```

que computa la función pedida.

Ej 2c pag. 77 (Davis): Dar un conjunto B infinito tal que $\Phi(b, b)\uparrow$ para sus elementos y que

$$f(x) = \begin{cases} 1 & \text{si } \Phi_x(x)\downarrow \\ 0 & \text{si } x \in B \\ \uparrow & \text{si no} \end{cases}$$

sea parcialmente computable.

Resolución: ¿Cuál es la diferencia con el ejercicio anterior? Ahora el conjunto debe ser infinito por lo tanto no podemos definir la función por extensión como antes y no podemos hacer un programa que compare caso por caso. Anteriormente lo hicimos porque era una manera facil de computar la pertenecia al conjunto. En este caso debemos dar un conjunto tal que podamos computar la pertenencia (o no-pertenencia) y que cumpla con la propiedad pedida.

El conjunto

$$B = \{\text{programas de la forma}\ldots\}$$

```
Z ← Z + 1 (con n repeticiones de esta operación, n > 0)
[A] IF Z ≠ 0 GOTO A
```

tiene infinitos programas que no terminan para toda entrada, en particular $\Phi(b, b) \uparrow$.

Ahora…, ¿Cómo podemos saber si $x \in B$? Dada la codificación de los programas no debería ser muy dificil saber si un programa tiene esa forma.

$$tiene_forma(x) = |x| > 1 \wedge muchas_sumas(x) \wedge ciclo(x)$$

donde

$$ciclo(x) = \begin{cases} 1 & \text{si } x[|x|] = \langle \#(A), \langle \#(A) + 2, \#(Z) - 1 \rangle \rangle \\ 0 & \text{si no} \end{cases}$$

se fija que la ultima instrucción el ciclo que queremos y

$$muchas_sumas(x) = \forall_{1 \leq i < |x|}\ x[i] = \langle 0, \langle 1, \#(Z) - 1 \rangle \rangle$$

se fija que todas las primeras instrucciones sean incrementos a Z. Con esto queda definida la función usando solamente funciones primitivas recursivas.

Ahora que tenemos a $tiene_forma(x)$ que decide si el programa x tiene esa forma. Entonces podemos escribir un programa en $\mathcal{S}$

```
IF tiene_forma(x) GOTO FIN
Y ← 1
Z ← Φx(x)
```

que computa la función pedida.

Ej 2d pag. 78 (Davis): Dar un conjunto B infinito tal que $\Phi(b, b) \uparrow$ para sus elementos y que

$$f(x) = \begin{cases} 1 & \text{si } \Phi_x(x) \downarrow \\ 0 & \text{si } x \in B \\ \uparrow & \text{si no} \end{cases}$$

sea **no** computable.

Resolución: Si tomamos el conjunto

$$B = \{p : \neg HALT(p, p)\}$$

todos sus *infinitos* elementos cumplen $\Phi(b, b) \uparrow$ trivialmente. Probemos que no es computable por el absurdo.

Supongamos $f(x)$ computable, ahora tomemos un índice de programa e genérico. La función $f(e) = 0$ si y solo si $e \in B$ pero por definición del conjunto entonces

$$f(e) = 0 \iff e \in B \iff \neg HALT(e, e)$$

en el otro caso, $f(e) = 1$ si y solo si $\Phi_e(e) \downarrow$ entonces

$$f(e) = 1 \iff \Phi_e(e) \downarrow \iff HALT(e, e)$$

Notemos que la función no debería indefinirse ya que el indice o bien pertenece al conjunto o bien $\Phi_e(e) \downarrow$. En definitiva, dado este conjunto B acabamos de ver que vale

$$f(x) = HALT(x, x)$$

como ya vimos que $HALT(x, x)$ no es computable llegamos a un absurdo. $\square$