Funciones computables

Franco Frizzo

30 de octubre de 2024

Índice

1. El lenguaje $\mathcal{S}$++

Intuitivamente, la noción de **función computable** es fácil de definir: una función $f : \mathbb{N}^n \to \mathbb{N}$ es computable si puede darse un algoritmo que la calcule para cualquier valor de entrada. Sin embargo, si queremos darle un poco más de precisión a esta definición, tenemos que determinar de manera precisa qué entendemos por "algoritmo".

Existen distintas formas de definir este concepto, de las cuales la más popular es la máquina de Turing. Descritas por primera vez por Alan Turing en un paper fundacional publicado en 1937 [1], se trata de un autómata sofisticado, que además de tener estados (como un autómata finito o de pila), cuenta con una cinta infinita que hace las veces de entrada y memoria de trabajo. Una diferencia fundamental entre las máquinas de Turing y otros autómatas que vimos en la materia es que además de aceptar o rechazar una entrada determinada, también pueden "colgarse" sin alcanzar nunca ninguno de los dos resultados.

La **tesis de Church-Turing** nos dice que este formalismo captura de manera completa lo que intuitivamente entendemos como "algoritmo"; es decir, que no existen funciones computables para las que no sea posible dar una máquina de Turing.

Existen también muchos otros formalismos que se han demostrado equivalentes a las máquinas de Turing (también llamados Turing-completos). En la práctica de la materia ya vimos uno

de ellos (las gramáticas sin restricciones o de tipo 3). Hoy trabajaremos con otro: un lenguaje de programación muy sencillo que llamaremos $\mathcal{S}$++.[1]

1.1. Variables de un programa

En un programa de $\mathcal{S}$++ se pueden usar **variables** de tres tipos:

- **Parámetros** o variables de entrada: $X_1, X_2, X_3, ...$
- **Variables locales**: $Z_1, Z_2, Z_3, ...$
- **Salida**: Y

1.2. Instrucciones

Sean V una variable y P un programa. Una **instrucción** de $\mathcal{S}$++ es una de las siguientes:

- **Incremento** (V++): Incrementa el valor de V en 1.
- **Decremento** (V−−): Decrementa el valor de V en 1 (si es 0, no hace nada).
- **Loop** (**while** $V \neq 0$ **do** { P }): Ejecuta P mientras V sea distinto de 0.
- **Instrucción vacía** (**pass**): No hace nada.

1.3. Programas

Un **programa** es una sucesión de instrucciones. Para hacerlos más legibles, las escribimos en líneas separadas. Por ejemplo:

P_1:
```
Z1++
Z1++
while Z1 ≠ 0 do {
    Y++
    Z1−−
}
```

1.4. Función computada

Dado un programa P, para cada $n \in \mathbb{N}$, se define $\Psi_P^{(n)} : \mathbb{N}^n \to \mathbb{N}$, la **función computada** por P con n entradas.

$\Psi_P^{(n)}$ es una función **parcial**, es decir, puede ser que para ciertos valores de entrada no esté definida. Dada una entrada $x_1, ..., x_n \in \mathbb{N}$, usamos la notación:

- $\Psi_P^{(n)}(x_1, ..., x_n) \downarrow$ si la función está definida.
- $\Psi_P^{(n)}(x_1, ..., x_n) \uparrow$ si la función no está definida.

Llamamos $\text{Dom}\left(\Psi_P^{(n)}\right) \subseteq \mathbb{N}^n$, el **dominio** de la función, al conjunto de valores para los que está definida, y decimos que la función es **total** si su dominio es todo el conjunto $\mathbb{N}^n$.

Para obtener el valor de la función $\Psi_P^{(n)}$ para los valores de entrada $x_1, ..., x_n \in \mathbb{N}$, seguimos estos pasos:

1. Inicializamos las variables $X_1, ...X_n$ en los valores $x_1, ..., x_n$, y las demás variables en 0.
2. Ejecutamos en orden las instrucciones de P.
3. Si la ejecución termina, entonces $\Psi_P^{(n)}(x_1, ..., x_n) \downarrow$ y su valor es el de la variable Y al finalizar la ejecución. Si la ejecución no termina, entonces $\Psi_P^{(n)}(x_1, ..., x_n) \uparrow$.

[1] Para convencerse de que $\mathcal{S}$++ es Turing-completo, consultar la clase teórica.

Por ejemplo, en el caso del programa P_1 definido antes (que no mira ninguna de las variables de entrada), podemos ver que para todo $n \in \mathbb{N}$, $x_1, ..., x_n \in \mathbb{N}$ vale que $\Psi_P^{(n)}(x_1, ..., x_n) = 2$. Es decir, la función computada por P_1 (para cualquier cantidad de entradas) es la función constante 2.

1.5. Macros

En $\mathcal{S}$++, por simplicidad, no existe la posibilidad de definir y hacer llamados a funciones. Sin embargo, dado lo minimal del lenguaje, muchas operaciones sencillas requieren una gran cantidad de instrucciones para ser llevadas a cabo.

Para evitar repetir siempre lo mismo y simplificar la escritura y lectura de programas, podemos definir **macros**. Una macro no es más que una sucesión de instrucciones para las cuales definimos una notación particular.

Por ejemplo, supongamos que en un programa queremos poner una variable V en 0. Esto podemos hacerlo combinando un loop con un decremento: mientras el valor de V no sea 0, reducimos dicho valor en 1. Por lo tanto, podemos definir la siguiente macro:

$V = 0$: **while** $V \neq 0$ **do** {
    $V{-}{-}$
}

Un programa definido mediante el uso de macros recibe el nombre de **pseudo-programa**. Todo pseudo-programa tiene un programa equivalente, que se obtiene **expandiendo** todas las macros que aparecen en el mismo, es decir, reemplazándolas por el código correspondiente. Por ejemplo, el pseudo-programa:

$Y = 0$
$Y{+}{+}$

que computa la función constante 1, se corresponde con el siguiente programa:

while $Y \neq 0$ **do** {
    $Y{-}{-}$
}
$Y{+}{+}$

En ocasiones puede ser necesario utilizar variables auxiliares dentro de una macro. En ese caso es importante indicar que dichas variables deben ser **frescas**, es decir, al expandir la macro deben ser reemplazadas por una variable del tipo Z_i que no haya sido usada en ningún otro lugar del programa (esto siempre es posible, porque existen infinitas variables del tipo Z_i). Al definir una macro, usaremos la convención $Z_n, Z'_n, Z''_n, ...$ para indicar las variables auxiliares que deben ser frescas.

Ejercicio 1. Dar una macro para la pseudo-instrucción

$$V_1 = V_2$$

que copia el valor de la variable V_2 a la variable V_1.

Solución.

$V_1 = V_2$:
// Ponemos V_1 en 0
$V_1 = 0$
// Pasamos el valor de V_2 a una variable auxiliar fresca
while $V_2 \neq 0$ **do** {
  $Z_n{+}{+}$
  $V_2{-}{-}$
}
// Pasamos el valor de la variable auxiliar a V_1 y restauramos V_2
while $Z_n \neq 0$ **do** {
  $V_1{+}{+}$
  $V_2{+}{+}$
  $Z_n{-}{-}$
}

Notar que en esta macro Z_n es una variable auxiliar fresca, mientras que V_1 y V_2 son "parámetros" de la macro que deben instanciarse en variables concretas al usarla dentro de un pseudo-programa.

Para el resto de la clase, consideraremos como dadas también a las siguientes macros, cuya definición se pide en el ejercicio 1 de la práctica 8:

$V = V + k$	Suma k al valor de la variable V
$V = k$	Asigna el valor k (constante) a la variable V
$V_1 = V_1 + V_2$	Suma el valor de la variable V_2 al valor de la variable V_1
if $V \neq 0$ **then** { P }	Si la variable V es distinta de 0, ejecuta el programa P
if $V \neq 0$ **then** { P_1 } **else** { P_2 }	Si la variable V es distinta de 0, ejecuta el programa P_1, y en caso contrario ejecuta el programa P_2
loop	Entra en un ciclo infinito
$V = \Psi_P^{(n)}(V_1, ..., V_n)$	Si el programa P termina al tomar como entrada los valores de $V_1, ..., V_n$, asigna el resultado de su ejecución a la variable V; en caso contrario, se cuelga

Ejercicio 2.

a. Exhibir un pseudo-programa P en el lenguaje $\mathcal{S}$++ que compute la función $* : \mathbb{N}^2 \to \mathbb{N}$ definida por $*(x, y) = x \cdot y$.

b. Sea Q el programa en $\mathcal{S}$++ obtenido a partir de P. Caracterizar las siguientes funciones:

i. $\Psi_Q^{(1)} : \mathbb{N} \to \mathbb{N}$ ii. $\Psi_Q^{(2)} : \mathbb{N}^2 \to \mathbb{N}$ iii. $\Psi_Q^{(3)} : \mathbb{N}^3 \to \mathbb{N}$

Solución.

a. Para resolver este ejercicio, vamos a asumir que contamos con las siguientes macros:
- $V_1 = V_2$, que asigna el valor de V_2 a la variable V_1,
- $V_1 = V_1 + V_2$, que suma a la variable V_1 el valor de la variable V_2.

La idea del programa es simplemente sumar X_2 veces el valor de X_1 a la variable Y:

$$
\begin{array}{ll}
P\text{:} & Z_1 = X_2 \\
& \textbf{while } Z_1 \neq 0 \textbf{ do } \{ \\
& \quad Y = Y + X_1 \\
& \quad Z_1{-}{-} \\
& \}
\end{array}
$$

El programa también sería válido si en vez de copiarse el valor de X_2 en la variable Z_1, se lo modicara directamente. Sin embargo, escribir programas que respeten sus entradas tiene la ventaja de que es más sencillo reutilizarlos como macros.

b.
i. $\Psi_Q^{(1)}(x)$ indica el valor computado por Q cuando la variable de entrada X_1 toma el valor x, y el resto de las variables de entrada el valor 0. Siguiendo el código, es sencillo ver que para todo x vale que $\Psi_Q^{(1)}(x) = 0$.

ii. $\Psi_Q^{(2)}(x, y)$ indica el valor computado por Q cuando las variables de entrada X_1 y X_2 toman los valores x, y respectivamente, y el resto de las variables de entrada el valor 0. En este caso, $\Psi_Q^{(2)}(x, y) = x \cdot y$.

iii. $\Psi_Q^{(3)}(x, y, z)$ indica el valor computado por Q cuando las variables de entrada X_1, X_2 y X_3 toman los valores x, y, z respectivamente, y el resto de las variables de entrada el valor 0. Como el valor de X_3 no se utiliza en el programa, el comportamiento es similar al caso anterior, es decir, $\Psi_Q^{(3)}(x, y, z) = x \cdot y$.

2. Funciones y lenguajes computables

2.1. Funciones computables y parcialmente computables

Dada una función $f : \mathbb{N}^n \to \mathbb{N}$, diremos que:

- f es $\mathcal{S}$++-parcialmente computable, o simplemente **parcialmente computable**, si existe algún programa P tal que $f = \Psi_P^{(n)}$.
- f es $\mathcal{S}$++-computable, o simplemente **computable**, si es parcialmente computable y además es *total*, es decir, $f(x_1, ..., x_n) \downarrow$ para cualesquiera $x_1, ..., x_n$.

Cabe señalar (y esto al principio puede ser un poco confuso) que todas las funciones computables son parcialmente computables (pero no al revés).

2.2. Composición de funciones computables

En general, si sabemos que una función $f : \mathbb{N}^n \to \mathbb{N}$ es parcialmente computable, será aceptable utilizar en nuestros programas la pseudo-instrucción:

$$V = f(V_1, ..., V_n)$$

cuyo efecto es:

- si f está definida para los valores de las variables $V_1, ..., V_n$, asigna su valor a la variable V,
- si f no está definida para estos valores, hace entrar al programa en un ciclo infinito.

Esto es posible porque, si f es parcialmente computable, debe existir un programa P tal que $f = \Psi_P^{(n)}$, y como vimos antes, podemos definir una macro que ejecute dicho programa y asigne el valor de su salida a la variable V.

2.3. Funciones primitivas recursivas

Es posible demostrar que toda función **primitiva recursiva** es **computable**. Esto es un ejercicio de la guía y los detalles de la demostración se los dejamos a ustedes, pero la idea general es usar inducción estructural: por un lado, mostramos que las funciones iniciales son computables (dando los programas en $\mathcal{S}$++ correspondientes), y por otro lado, mostramos que las dos formas posibles de combinar funciones primitivas recursivas (composición y recursión primitiva) pueden llevarse a cabo también en $\mathcal{S}$++ combinando programas.

Este resultado nos brinda una enorme paleta de funciones que ya sabemos computables, y que podemos usar sin problemas a la hora de definir programas en $\mathcal{S}$++.

2.4. Lenguajes computables

Consideremos un alfabeto Σ, y un lenguaje $\mathcal{L} \subseteq \Sigma^*$. Decimos que $\mathcal{L}$ es un **lenguaje computable** si, considerando una codificación de cadenas $\Sigma^* \to \mathbb{N}$ (por ejemplo, la función ρ_Σ definida en el ejercicio 13 de la práctica 7), su predicado característico

$$p_{\mathcal{L}}(x) = \begin{cases} 1 & \text{si la cadena codificada por } x \text{ pertenece a } \mathcal{L} \\ 0 & \text{si no} \end{cases}$$

es computable.

Ejercicio 3. Demostrar que el siguiente lenguaje es computable:

$$\mathcal{L} = \{a^n b^{2n} \mid n \geq 0\}.$$

Solución. Como vimos en la práctica 7, las funciones cabeza_Σ y cola_Σ, que nos permiten separar una cadena en su primer carácter y el resto, son primitivas recursivas. Usando estas funciones, podemos definir un programa que itere sobre los caracteres de una cadena:

$Z_1 = X_1$
$Z_2 = \text{cabeza}_\Sigma(Z_1)$
while $Z_2 == \#_\Sigma(a)$ **do** {
    $Z_3{+}{+}$
    $Z_1 = \text{cola}_\Sigma(Z_1)$
    $Z_2 = \text{cabeza}_\Sigma(Z_1)$
}
while $Z_2 == \#_\Sigma(b)$ **do** {
    $Z_4{+}{+}$
    $Z_1 = \text{cola}_\Sigma(Z_1)$
    $Z_2 = \text{cabeza}_\Sigma(Z_1)$
}
if $Z_1 == \rho_\Sigma(\lambda) \wedge 2 \cdot Z_3 = Z_4$ **then** {
    $Y = 1$
}

El primer ciclo recorre la cadena mientras encuentre el carácter a, y va contando las apariciones en la variable Z_3. El segundo ciclo hace lo propio con el carácter b en la variable Z_4. Por último, se verifica que se hayan recorrido todos los caracteres de la cadena y que se

cumpla la igualdad pedida, es decir, que la cantidad de apariciones de b sea el doble que la de apariciones de a. Solo en este caso, se almacena el valor 1 en la variable Y, convirtiéndose en la salida del programa. En caso contrario, se devuelve el valor por defecto de la variable Y, que es 0.

Solución alternativa. Otra posibilidad para llegar a una demostración es combinar los conceptos aprendidos en guías anteriores con los resultados teóricos que aparecen más arriba.

Por ejemplo, en la guía 6 vimos cómo definir gramáticas libres de contexto. Para el lenguaje de este ejercicio, podemos generarlo con la gramática $G = \langle V_N, V_T, P, S\rangle$, donde P:

$$S \rightarrow aSbb \mid \lambda$$

Claramente, la gramática G no es recursiva a izquierda, y en el ejercicio 15 de la guía 7 vimos que los lenguajes generados por gramáticas libres de contexto que no sean recursivas a izquierda son primitivos recursivos. Como además, sabemos que todas las funciones primitivas recursivas son computables, concluimos que necesariamente el lenguaje $\mathcal{L}$ debe ser computable.

3. Codificación de programas

En la clase anterior, vimos que podemos **codificar** distintos tipos de elementos como números naturales, poniéndolos en biyección con el conjunto $\mathbb{N}$. Vimos también que nos interesa encontrar codificaciones tales que sus observadores básicos, y otras operaciones comunes, resulten primitivas recursivas.

Dado que los programas de $\mathcal{S}$++ son estructuras finitas, también podemos definir formas de codificarlos como números naturales. Más aún, podemos dar una codificación canónica utilizando las funciones codificadoras de pares y de listas que vimos en la clase anterior.

Comenzaremos por dar una forma de codificar las instrucciones. Si I es una instrucción de $\mathcal{S}$++, definimos su código $\#(I)$ como:

$$\#(I) = \begin{cases} \langle a, b\rangle + 1 & \text{si } I \neq \textbf{pass} \\ 0 & \text{si no} \end{cases}$$

donde $\langle \bullet, \bullet\rangle : \mathbb{N}^2 \to \mathbb{N}$ es la función codificadora de pares, y:

- $a = \#(V)$, siendo V la variable mencionada en I, de forma que:
  - $\#(Y) = 0$
  - $\#(X_i) = 2i - 1$
  - $\#(Z_i) = 2i$
- el valor de b es:
  - 0 si I es un incremento.
  - 1 si I es un decremento.
  - $\#(P) + 2$ si I es un loop cuyo cuerpo es P.

Como los programas no son otra cosa que listas de instrucciones, podemos codificarlos usando la función codificadora de listas. Dado P, un programa de $\mathcal{S}$++, definimos su número de Gödel $\#(P)$ de la siguiente forma:

$$\#(P) = [\#(I_1), \#(I_2), ..., \#(I_k)]$$

donde $[\bullet, ..., \bullet] : \mathbb{N}^* \to \mathbb{N}$ es la función codificadora de listas, y $I_1, I_2, ..., I_k$ son las instrucciones de P.

Dado $e \in \mathbb{N}$, usamos la notación $\Phi_e^{(n)} : \mathbb{N}^n \to \mathbb{N}$ para denotar la función computada por el programa cuyo número es e, con n entradas. Es importante confundir las notaciones $\Psi^{(n)}$ y $\Phi^{(n)}$. Si $e = \#(P)$, entonces $\Phi_e^{(n)} = \Psi_P^{(n)}$.

Ejercicio 4. Decimos que un programa es *peligroso* cuando contiene alguna instrucción de la forma **while** $V \neq 0$ **do** { P } (donde V es una variable y P un programa) tal que el programa P no contiene la instrucción $V{-}{-}$, ni de forma directa ni dentro del cuerpo de un loop.

Demostrar que el siguiente predicado es primitivo recursivo:

$$p(x) = \begin{cases} 1 & \text{si el programa cuyo número es } x \text{ es peligroso} \\ 0 & \text{si no.} \end{cases}$$

Solución. La idea del ejercicio es "decodificar" el programa cuyo número el predicado recibe por parámetro, para así poder analizar sus instrucciones. Sabemos que si el programa con número x consiste en las instrucciones $I_1, I_2, ..., I_k$, entonces x satisface

$$x = [\#(I_1), \#(I_2), ..., \#(I_k)].$$

Podríamos resolver el problema analizando los códigos de cada una de estas instrucciones y verificando si alguna de ellas cumple la condición que hace peligroso al programa.

Digamos que una *instrucción* es peligrosa si es un loop tal que, si V es la variable que aparece en la condición, la instrucción $V{-}{-}$ no aparece en el cuerpo. Si pudiéramos demostrar que el predicado

$$\tilde{p}(x) = \begin{cases} 1 & \text{si la instrucción codificada por } x \text{ es peligrosa} \\ 0 & \text{si no} \end{cases}$$

es primitivo recursivo, podríamos reescribir p como:

$$p(x) = \exists_{1 \leq t \leq |x|}(\tilde{p}(x[t])).$$

Al tratarse de la composición de un existencial acotado y funciones que ya demostramos (en la guía anterior) que son primitivas recursivas (la función observadora de listas $\bullet\,[\bullet]$ y su longitud $|\bullet|$), el hecho de que $\tilde{p}$ sea primitivo recursivo implica directamente que p también lo es.

Analicemos, entonces, el predicado $\tilde{p}$ con más detalle. Sabemos que las instrucciones de un programa se codifican de la forma

$$\langle a, b \rangle + 1$$

donde a codifica la variable mencionada en la instrucción y b el tipo de instrucción.

Esto quiere decir que, si $x > 0$ es el código de una instrucción, podemos conocer su tipo mirando el valor de b (mediante la función observadora $r(x-1)$), y cuál es la variable mencionada mirando el valor de a (mediante la función observadora $l(x-1)$). Para simplificar la notación, vamos a definir las funciones auxiliares (primitivas recursivas):

- $\text{tipo}(x) = r(x-1)$, que nos da el tipo de la instrucción codificada por x,

- $\text{var}(x) = l(x-1)$, que nos da el código de la instrucción codificada por x.

Para saber si la instrucción es un loop, basta con mirar el valor de $\text{tipo}(x)$. Sabemos que la instrucción es de la forma

while $V \neq 0$ **do** $\{$
$\quad P$
$\}$

si $x > 0 \wedge \text{tipo}(x) \geq 2$.

Resta comprobar que la variable V no aparezca mencionada en ninguna de las instrucciones de P que sea un decremento. Por un lado, podemos obtener el código de V como en cuyo caso $\#(V) = \text{var}(x)$. La codificación del cuerpo del loop, por su parte, es $\#(P) = \text{tipo}(x) - 2$, lo cual motiva definir una nueva función auxiliar:

$$\text{cuerpo}(x) = \text{tipo}(x) - 2,$$

que nos devuelve la codificación del cuerpo de un loop.

Usando todo lo anterior podemos definir el siguiente predicado:

$$\text{decrementa}(x, y) = \begin{cases} & \text{si la instrucción codificada por } x \text{ es } V{-}{-} \\ 1 & \text{o un loop cuyo cuerpo contiene la instrucción } V{-}{-}, \\ & \text{siendo } y = \#(V) \\ 0 & \text{si no} \end{cases}$$

Este predicado es primitivo recursivo, ya que podemos escribirlo como:

$$\text{decrementa}(x, y) = x > 0 \wedge \Big(\Big(\text{tipo}(x) == 1 \wedge \text{var}(x) == y \Big) \vee \Big(\text{tipo}(x) \geq 2 \wedge \exists_{1 \leq t \leq |\text{cuerpo}(x)|} \Big(\text{decrementa}\Big(\text{cuerpo}(x)[t], y \Big) \Big) \Big) \Big).$$

Aquí tenemos una combinación de funciones primitivas recursivas que utiliza composición y recursión *global*,[2] ya que para definir $\text{decrementa}(x, y)$ utilizamos los valores de $\text{decrementa}(\text{cuerpo}(x)[t], y)$ para $t = 1, ..., |\text{cuerpo}(x)|$ y por cómo están definidas la codificaciones de pares y de listas, sabemos que

$$\text{cuerpo}(x)[t] = (r(x-1) - 2)[t] < x,$$

es decir, el argumento de la llamada recursiva del predicado es estrictamente menor que el valor para el cual que se lo está queriendo definir.[3]

Combinando todo lo anterior, podemos escribir al predicado $\tilde{p}$ de la siguiente forma:

$$\tilde{p}(x) = x > 0 \wedge \text{tipo}(x) \geq 2 \wedge \neg \Big(\exists_{1 \leq t \leq |\text{cuerpo}(x)|} \Big(\text{decrementa}\Big(\text{cuerpo}(x)[t], \text{var}(x) \Big) \Big) \Big)$$

donde verificamos que la instrucción codificada por x sea un loop $(x > 0 \wedge \text{tipo}(x) \geq 2)$ y, usando el predicado definido antes, que no exista una instrucción en su cuerpo que decremente (de forma directa o indirecta) la variable cuyo código es $\text{var}(x)$. Con argumentos

[2] Llamamos recursión global a una forma de recursión donde para definir el valor de $f(t, x_1, ..., x_n)$ puede usarse cualquiera de los valores $f(0, x_1, ..., x_n), f(1, x_1, ..., x_n), ..., f(t-1, x_1, ..., x_n)$. Damos aquí por demostrado (es un ejercicio interesante) que toda función que puede definirse utilizando recursión global puede definirse también utilizando recursión primitiva.

[3] Para convencerse de esto, tener en cuenta que para todo x, t, se cumple que $r(x) \leq x$ y $x[t] \leq x$.

similares a los que usamos antes, esta escritura muestra que $\tilde{p}$ es primitivo recursivo, concluyendo la demostración.

4. Snap, Step y programa universal

4.1. Configuración instantánea y la función Snap

Dado un programa P, $n \in \mathbb{N}$, y valores $x_1, ..., x_n \in \mathbb{N}$ para las variables de entrada, podemos pensar en la ejecución del mismo como una serie de **pasos de cómputo**. En cada paso, se ejecuta una instrucción y se modifica el estado del cómputo. Dado un momento del tiempo, podemos representar el estado del cómputo mediante la **configuración instantánea** del programa, que tiene dos componentes fundamentales:

1. La **instrucción** que toca ejecutar en el próximo paso. La representamos como una lista $[i_1, ..., i_k]$ donde:
   - i_1 es el índice de la instrucción de P que toca ejecutar a continuación,
   - si la instrucción i_1-ésima de P es un loop, i_2 es el índice de la instrucción *dentro del loop* que toca ejecutar a continuación, y así sucesivamente.

   Por ejemplo, en el siguiente programa

   $$\begin{array}{l} Z_1{+}{+} \\ \textbf{while } X_1 \neq 0 \textbf{ do } \{ \\ \quad Z_1{+}{+} \\ \quad X_1{-}{-} \\ \} \end{array}$$

   si la próxima instrucción a ejecutar es $X_1{-}{-}$ dentro del loop, esto será indicado mediante la lista $[2, 2]$.

   Si la ejecución del programa ya terminó, entonces este valor será una lista con un único elemento, y este elemento será la cantidad de instrucciones del programa más uno.
2. Los **valores** de cada una de las variables. Estos también serán representados mediante una lista, en el orden $Y, X_1, Z_1, X_2, Z_2, X_3, Z_3, ...$, hasta la última variable cuyo valor sea distinto de cero.

De esta forma, la configuración instantánea de un programa puede pensarse como un par de números naturales $\langle a, b \rangle$, donde a es la lista que representa la próxima instrucción a ejecutar y b la lista de los valores de las variables. Es decir, la configuración instantánea es representable como un número natural mediante las funciones codificadoras que ya conocemos.

Definimos la función $\text{Snap}^{(n)} : \mathbb{N}^{n+2} \to \mathbb{N}$ de modo que si $e = \#(P)$, entonces $\text{Snap}^{(n)}(t, e, x_1, ..., x_n)$ codifica la configuración instantánea del programa P, con entradas $x_1, ..., x_n$, transcurridos t pasos de su ejecución. Es posible demostrar (no lo haremos acá) que, para todo $n \in \mathbb{N}$, la función $\text{Snap}^{(n)}$ es primitiva recursiva.

4.2. El predicado Step

Definimos el predicado $\text{Step}^{(n)} : \mathbb{N}^{n+2} \to \{0, 1\}$ de forma tal que, si $e = \#(P)$, entonces:

$$\text{Step}^{(n)}(t, e, x_1, ..., x_n) = \begin{cases} 1 & \text{si la ejecución de } P \text{ con entradas } x_1, ..., x_n \\ & \text{finaliza tras } t \text{ o menos pasos de ejecución} \\ 0 & \text{en caso contrario.} \end{cases}$$

El valor de $\text{Step}^{(n)}$ puede ser obtenido a partir de $\text{Snap}^{(n)}$ de una forma bastante sencilla[4] y, por lo tanto, se puede mostrar que también es primitivo recursivo.

4.3. Programa universal

Contando con los predicados $\text{Snap}^{(n)}$ y $\text{Step}^{(n)}$, podemos definir un programa en $\mathcal{S}$++ que, dado el número de otro programa, **simule** la ejecución del mismo para determinadas entradas. Escrito en forma de pseudo-programa, el mismo podría ser de la pinta:

$\Phi^{(n)}_{X_1}(X_2, ..., X_{n+1})$:

$Z_2 = 1$
while $Z_2 \neq 0$ **do** {
  // Z_2 es 0 si la ejecución ya terminó y 1 si no
  $Z_2 = \neg\text{Step}^{(n)}(Z_1, X_1, X_2, ..., X_{n+1})$
  // Z_1 contiene la cantidad de pasos de ejecución transcurridos
  Z_1++
}
$Y = r\Big(\text{Snap}^{(n)}(Z_1, X_1, X_2, ..., X_{n+1})\Big)[1]$

Las consecuencias de que sea posible hacer esto son sumamente interesantes y las seguiremos explorando en profundidad en la próxima clase.

Ejercicio 5. Demostrar que la siguiente función es parcialmente computable.

$$f(x, y, z) = \begin{cases} 1 & \text{si la ejecución de } \Phi^{(1)}_x(z) \text{ termina estrictamente en} \\ & \text{menos pasos que la ejecución de } \Phi^{(1)}_y(z) \\ \uparrow & \text{en caso contrario.} \end{cases}$$

Aclaración: Si para determinada entrada un programa no termina, consideramos que su ejecución tiene infinitos pasos.

Solución. Construyamos un programa en $\mathcal{S}$++ que compute f. La idea es utilizar el predicado $\text{Step}^{(1)}$ para "seguir el rastro" del programa cuyo número es x, y así descubrir cuántos pasos tarda en terminar de ejecutar, si es que en algún momento lo hace. Luego, podemos usar nuevamente $\text{Step}^{(1)}$ para averiguar si el programa con número y termina de ejecutar en una cantidad menor o igual de pasos. Nuestro programa debe devolver 1 si esto último resulta ser falso, y colgarse en caso contrario.

Como el predicado $\text{Step}^{(1)}$ es primitivo recursivo, también es computable, así que podemos usarlo directamente dentro de nuestros programas (por supuesto, teniendo presente que en rigor se trata de una macro que debería ser expandida para obtener un programa válido).

Un posible programa en $\mathcal{S}$++ es el siguiente:

[4] ¿Cómo? Pensarlo un momento.

P: **while** $Z_1 == 0$ **do** {
$Z_1 = \text{Step}^{(1)}(Z_2, X_1, X_3)$
$Z_2{+}{+}$
}
$Z_2{-}{-}$
$Z_3 = \text{Step}^{(1)}(Z_2, X_2, X_3)$
if $Z_3 \neq 0$ **then** {
loop
} **else** {
$Y = 1$
}

El primer ciclo del programa va incrementando el valor de Z_2 hasta encontrar una cantidad de pasos tras la cual el programa X_1 termine para la entrada X_3. Si el programa X_1 se cuelga para la entrada X_3, este ciclo nunca termina, lo cual es correcto. En cambio, si el ciclo termina, el valor final de Z_2 es la cantidad de pasos tras los cuales el programa termina más uno. Por eso, acto seguido, decrementamos Z_2 para obtener la cantidad de pasos exacta.

Acto seguido, usamos nuevamente $\text{Step}^{(1)}$ para verificar si tras esta cantidad de pasos, el programa X_2 también terminó para la entrada X_3. En caso afirmativo, el valor de Z_3 pasará a ser 1, con lo cual nuestro programa entrará en un loop infinito; de lo contrario, finalizaremos la ejecución devolviendo el valor 1.

Solución alternativa. Otra posibilidad, en vez de dar explícitamente un programa en $\mathcal{S}{+}{+}$, es demostrar primero un resultado un poco más general, que nos será útil en otros ejercicios de este tipo. Se trata de la siguiente afirmación: si $p : \mathbb{N}^n \to \{0, 1\}$ es un predicado computable, entonces el *existencial no acotado* sobre este predicado:

$$\exists_t(p(t, x_1, ..., x_n)) = \begin{cases} 1 & \text{si existe } t \text{ tal que } p(t, x_1, ..., x_n) = 1 \\ \uparrow & \text{en caso contrario.} \end{cases}$$

es una función parcialmente computable.

Podemos demostrar esta afirmación escribiendo un sencillo programa en $\mathcal{S}{+}{+}$ que vaya recorriendo los valores posibles de t y evaluando para cada uno de ellos el predicado.

while $Z_1 == 0$ **do** {
$Z_1 = p(X_1, ..., X_n, Z_2)$
$Z_2{+}{+}$
}
$Y = 1$

Es importante tener en cuenta que, para que este programa sea válido, el predicado p debe ser total. En caso contrario, la ejecución podría quedar atrapada en el cómputo de p para algún valor de t e indenir el existencial, incluso aunque p sea verdadero para otro valor de t que no todavía no hayamos vericado.

Una vez que contamos con el existencial, si encontramos un predicado apropiado, podemos usarlo para reescribir f. Al igual que en la solución anterior, intentemos aprovechar el predicado $\text{Step}^{(1)}$, que ya sabemos primitivo recursivo. Las siguientes son dos condiciones necesarias para que $f(x, y, z) = 1$:

(i) El programa con número x termina para la entrada z. Es decir, debe existir algún $t \in \mathbb{N}$ tal que $\text{Step}^{(1)}(t, x, z)$ sea verdadero.

(ii) El programa con número y, para la entrada z, no termina antes ni al mismo tiempo que el programa x para la misma entrada. Equivalentemente, debe existir algún tiempo t para el cual el programa x ya haya terminado, pero el programa z todavía no; o, más formalmente, existe algún $t \in \mathbb{N}$ que cumple $\text{Step}^{(1)}(t, x, z) \wedge \neg\text{Step}^{(1)}(t, y, z)$.

Dado que (ii) implica (i), solo tendremos en cuenta la última condición.

Es sencillo advertir que (ii) también es condición suciente para que $f(x, y, z) = 1$. En caso contrario, o bien la ejecución de $\Phi_x^{(1)}(z)$ no termina o, para todo t tal que la ejecución de $\Phi_x^{(1)}(z)$ termina en t o menos pasos, la ejecución de $\Phi_y^{(1)}(z)$ también termina en t o menos pasos. En cualquiera de estas dos situaciones, $f(x, y, z) \uparrow$.

Por lo tanto, podemos reescribir

$$f(x, y, z) = \exists_t \Big(\text{Step}^{(1)}(t, x, z) \wedge \text{Step}^{(1)}(t, y, z) \Big)$$

donde el predicado al que hace referencia el existencial es primitivo recursivo y, por lo tanto, computable (total), de donde se sigue que f es parcialmente computable.

Referencias

[1] A. Turing, «On Computable Numbers, with an Application to the Entscheidungsproblem», *Proceedings of the London Mathematical Society*, n.º 1, pp. 230-265, 1937, [En línea]. Disponible en: https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

[2] J. E. Hopcroft, R. Motwani, y J. D. Ullman, «Chapter 8: Introduction to Turing Machines», en *Introduction to Automata Theory, Languages, and Computation*, 3.ª ed., Addison-Wesley, 2006, pp. 315-376.

[3] M. Davis, R. Sigal, y E. J. Weyuker, «Chapter 2: Programs and Computable Functions», en *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*, 2.ª ed., Academic Press, 1994, pp. 17-38.

[4] M. Davis, R. Sigal, y E. J. Weyuker, «Chapter 4: A Universal Program», en *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*, 2.ª ed., Academic Press, 1994, pp. 65-112.