Funciones $\mathcal{S}$-computables

Lógica y Computabilidad

Franco Frizzo

1 de septiembre de 2017

1. Programas y macros

Ejercicio 1

(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) ¿Qué es necesario hacer para obtener un programa en $\mathcal{S}$ a partir de P? ¿Qué hay que tener en cuenta al hacerlo?

(c) 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. $\Psi_Q^{(3)} : \mathbb{N}^3 \to \mathbb{N}$

Resolución

(a) Para resolver este ejercicio, vamos a asumir que contamos con macros que nos permiten asignarle a una variable el valor de otra, dar saltos incondicionales y sumar el contenido de dos variables[1]. Tenemos que escribir un pseudo-programa que multiplique los valores de X_1 y X_2; una idea sencilla es sumar el valor de X_1 un total de X_2 veces. De esta forma obtenemos el siguiente pseudo-programa:

$$
\begin{array}{lll}
P: & & Z \leftarrow X_2 \\
& [B] & \text{IF } Z \neq 0 \text{ GOTO } A \\
& & \text{GOTO } E \\
& [A] & Y \leftarrow Y + X_1 \\
& & Z \leftarrow Z \dot{-} 1 \\
& & \text{GOTO } B
\end{array}
$$

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

(b) P es un pseudo-programa, en lugar de ser verdaderamente un programa, porque utiliza macros para algunas operaciones que ya sabemos que son $\mathcal{S}$-computables. Para convertirlo en un programa es necesario *expandir* estas macros, es decir, reemplazarlas por las instrucciones que están abreviando. Al hacerlo, hay que tener en cuenta que:

[1] Ver la teórica y el ejercicio 1 de la práctica 2.

- Las variables utilizadas en la macro deben ser reemplazadas por variables que no hayan sido utilizadas en el programa principal (variables "*frescas*").
- Las etiquetas que aparezcan en saltos condicionales en el código de la macro, pero que no referencien a ninguna instrucción del mismo, deben ser reemplazadas por una etiqueta L referenciando a la instrucción inmediatamente debajo de la macro, siempre y cuando exista tal instrucción.
- Todas las demás etiquetas utilizadas en la macro deben ser reemplazadas por etiquetas que no ocurran en el programa principal.
- Si la macro computa $Y \leftarrow f(x)$, y en nuestro programa usamos $V \leftarrow f(x)$, durante la expansión debemos reemplazar las apariciones de Y por V en el código de la macro.
- Si la macro computa $f(X_1, \ldots, X_n)$ y queremos usar $f(V_1, \ldots, V_n)$, debemos reemplazar las apariciones de $X_1, \ldots, X_n$ por $V_1, \ldots, V_n$.

(c) I. $\Psi_Q^{(1)}(x)$ indica el valor computado por Q cuando la variable de entrada X_1 toma el valor x, y todas las demás entradas, el valor 0. Siguiendo el código, es sencillo ver 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 e y respectivamente, y todas las demás entradas, 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 y z respectivamente, y todas las demás entradas, el valor 0. Como el valor de X_3 no se utiliza en el programa, el comportamiento es similar al caso anterior, y tenemos que $\Psi_Q^{(3)}(x, y, z) = x \cdot y$.

2. Cómputos

Ejercicio 2

Sea P un programa que contiene exactamente una instrucción de la forma

$$Y \leftarrow Y + 1$$

(etiquetada o no).

Demostrar que para todo $x \in \mathbb{N}$, si $\Psi_P^{(1)}(x) \downarrow$, entonces al ejecutar P con x como única entrada se pasa por dicha instrucción por lo menos $\Psi_P^{(1)}(x)$ veces.

Resolución

Intuitivamente, podemos darnos cuenta de por qué la afirmación es cierta. Por un lado, sabemos que $\Psi_P^{(1)}(x)$, el resultado de la ejecución del programa P, estará almacenado en la variable Y al finalizar dicha ejecución. Además, esta variable comienza valiendo 0 y solo puede ser incrementada en uno por una instrucción de la forma

$$Y \leftarrow Y + 1.$$

Como solo hay una instrucción de este tipo en el programa, esta instrucción es la única responsable de llevar la variable Y al valor $\Psi_P^{(1)}(x)$. Por lo tanto, tendrá que ser ejecutada al menos $\Psi_P^{(1)}(x)$ veces.

Si bien lo anterior es perfectamente válido como argumento informal, no constituye una demostración. Necesitamos formalizar un poco más nuestro razonamiento, y para hacerlo, tendremos que recurrir a las herramientas con las que está rigurosamente definida la *semántica* del lenguaje $\mathcal{S}$: estados, descripciones instantáneas y cómputos.

Recordemos que las *descripciones instantáneas* son una representación completa de cómo se encuentra la ejecución de un programa en un momento dado. El resultado del cómputo de un programa se define como el valor que tiene la variable Y según la última de sus descripciones instantáneas. A su vez, cada descripción instantánea del cómputo está definida a partir de la anterior, partiendo desde una *descripción inicial* determinada por la misma semántica del lenguaje (las variables $X_1, X_2, \ldots$ contienen los valores de entrada, las demás variables comienzan en 0, etc.). Esto nos da una *estructura inductiva* para hacer razonamientos sobre la ejecución de los programas: si logramos demostrar que cierta propiedad vale para la descripción inicial, y que se preserva al pasar de una descripción instantánea a la siguiente, habremos probado que vale al finalizar la ejecución.

En nuestro caso, queremos probar que, al terminar de ejecutar el programa, la cantidad de veces que se pasó por la instrucción que incrementa la variable Y es mayor o igual que el valor que tiene la variable Y en ese momento. Es importante notar que esta propiedad se puede generalizar a cualquier instante de la ejecución: el valor de la variable Y *nunca* puede superar la cantidad de veces que fue incrementada. Tratemos de llevar esto al esquema inductivo que acabamos de esbozar.

Dado cualquier $x \in \mathbb{N}$ tal que $\Psi_P^{(1)}(x) \downarrow$, sea

$$d_0, \ldots, d_n$$

el cómputo del programa P a partir de la entrada $x \in \mathbb{N}$ ($d_0 = (1, \sigma_0)$ es la descripción inicial correspondiente). Además, en adelante, llamaremos

- m al índice de la instrucción de P que incrementa la variable Y.
- k_i (para $i = 0, \ldots, n$) a la cantidad de veces que se pasó por la m-ésima instrucción de P tras i pasos de su ejecución.
- $d_i[Y]$ al valor de Y indicado por la descripción d_i.

Lo que queremos demostrar es que, para todo $i = 0, \ldots, n$, se cumple $k_i \geqslant d_i[Y]$.

(C.B.) Por definición de estado inicial, $d_0[Y] = 0$. También, claramente, $k_0 = 0$. Así, tenemos que $k_0 \geqslant d_0[Y]$.

(P.I.) Supongamos que vale que $k_i \geqslant d_i[Y]$, para $i \in \{0, \ldots, n-1\}$. Queremos ver que también $k_{i+1} \geqslant d_{i+1}[Y]$.

Sabemos que $d_i = (j_i, \sigma_i)$, donde j_i es el índice de la próxima instrucción a ejecutar. Analicemos los casos posibles.

I. Si $j_i = m$, entonces
$$k_{i+1} = k_i + 1 \geqslant d_i[Y] + 1 = d_{i+1}[Y],$$
porque tanto el valor de la variable Y como la cantidad de veces que se ejecutó la m-ésima instrucción se incrementan en 1.

II. Si $j_i \neq m$, puede ser que la instrucción que se va a ejecutar directamente no involucre a la variable Y, que la involucre pero sea un salto condicional (que no modifica su valor), o bien que decremente su valor en 1. En los tres casos, $d_i[Y] \geqslant d_{i+1}[Y]$. Como además $k_{i+1} = k_i$, tenemos que
$$k_{i+1} = k_i \geqslant d_i[Y] \geqslant d_{i+1}[Y].$$

Lo anterior demuestra que la propiedad que planteamos se preserva a lo largo de todos los pasos del programa, y en particular, que vale al final de la ejecución, que es lo que inicialmente queríamos probar.

3. Codificación de programas

Ejercicio 3

Decimos que un programa P *tiene un salto al final* si posee alguna instrucción del tipo

$$\text{IF } V \neq 0 \text{ GOTO } L$$

tal que ninguna de las instrucciones de P está etiquetada con L.

Demostrar que el siguiente predicado es primitivo recursivo.

$$r_1(x) = \begin{cases} 1 & \text{si el programa cuyo número es } x \text{ tiene un salto al final} \\ 0 & \text{si no} \end{cases}$$

Resolución

La idea del ejercicio es "decodificar" el programa cuyo número recibimos por parámetro, para así poder analizar sus instrucciones. Sabemos que si el programa con número x consiste en las instrucciones $I_1, I_2, I_3, \ldots, I_k$, entonces x satisface

$$x + 1 = [\#(I_1), \#(I_2), \#(I_3) \ldots, \#(I_k)]$$

Podríamos resolver el problema analizando los códigos de cada una de estas instrucciones y verificando si algunas de ellas se trata de un *salto al final*. Más aún, si pudiéramos demostrar que el predicado

$$\tilde{r}_1(x, y) = \begin{cases} 1 & \text{si la instrucción codificada por } y \text{ es un salto al final en el programa de número } x \\ 0 & \text{si no} \end{cases}$$

es primitivo recursivo, podríamos reescribir r_1 como

$$r_1(x) = (\exists t)_{\leqslant |x+1|} \Big(t > 0 \wedge \tilde{r}_1\big(x, (x+1)[t]\big) \Big)$$

Como todas las demás funciones que aparecen en la composición (el existencial acotado, los observadores de listas, etc.) son primitivas recursivas, de esta manera quedaría probado que r_1 es primitivo recursivo.

Analicemos, entonces, el predicado $\tilde{r}_1$ con más detalle. Sabemos que las instrucciones de un programa se codifican en la forma

$$\langle a, \langle b, c \rangle \rangle$$

donde a codifica la etiqueta de la instrucción, b codifica el tipo de instrucción y c, la variable involucrada.

Para saber si una instrucción es un salto al final, alcanza con conocer el valor de b, que podemos obtener a partir de una instrucción y como $l(r(y))$. Sabemos que la instrucción y es del tipo

$$\text{IF } V \neq 0 \text{ GOTO } L$$

si y solo si $l(r(y)) \geqslant 3$, y que, en tal caso, $\#(L) = l(r(y)) - 2$.

Además, debemos verificar si la etiqueta L en cuestión aparece o no en alguna instrucción del programa. Podemos hacerlo inspeccionando el valor de a en la codificación de cada una de las instrucciones; este valor se obtiene, para cada instrucción $\tilde{y}$, como $l(\tilde{y})$.

En conclusión, el predicado $\tilde{r}_1$ puede escribirse como

$$\tilde{r}_1(x, y) = l(r(y)) \geqslant 3 \wedge \neg(\exists t)_{\leqslant |x+1|} \Big(t > 0 \wedge l\big((x+1)[t]\big) = l(r(y)) \dot{-} 2 \Big)$$

que, *en criollo*, dice algo así como "*la instrucción codificada por y es un salto, y no existe ningún t tal que la t-ésima instrucción del programa cuyo número es x tiene la misma etiqueta a la que apunta el salto codificado por y*". De nuevo, no es difícil ver que este predicado es primitivo recursivo, lo cual concluye la demostración.

Ejercicio adicional

Decimos que un programa P *respeta sus entradas* si no tiene instrucciones de los tipos

$$X_i \leftarrow X_i + 1 \qquad \text{o} \qquad X_i \leftarrow X_i \dot{-} 1$$

para ningún $i \in \mathbb{N}$.

Demostrar que el siguiente predicado es primitivo recursivo.

$$r_2(x) = \begin{cases} 1 & \text{si el programa cuyo número es } x \text{ respeta sus entradas} \\ 0 & \text{si no} \end{cases}$$

Resolución

La estructura general de la resolución es muy similar a la del ejercicio anterior. En primer lugar, decimos que una instrucción del lenguaje $\mathcal{S}$ *respeta las entradas* si no es de los tipos

$$X_i \leftarrow X_i + 1 \qquad \text{o} \qquad X_i \leftarrow X_i \dot{-} 1$$

para ningún i, y definimos el predicado

$$\tilde{r}_2(y) = \begin{cases} 1 & \text{si la instrucción codificada por } y \text{ respeta las entradas} \\ 0 & \text{si no} \end{cases}$$

Esta vez, no necesitamos que el predicado tenga al programa como uno de sus parámetros, ya que la propiedad de *respetar las entradas* depende exclusivamente de la instrucción en cuestión.

Análogamente al ejercicio anterior, si pudiéramos demostrar que el predicado $\tilde{r}_2$ es primitivo recursivo, el problema estaría resuelto, ya que reescribiendo r_2 como

$$r_2(x) = (\forall t)_{\leqslant |x+1|} \Big(t = 0 \vee \tilde{r}_2\big((x+1)[t]\big) \Big)$$

resultaría evidente que este predicado es también primitivo recursivo.

En este caso, para determinar si una instrucción y codificada en la forma

$$y = \langle a, \langle b, c \rangle \rangle$$

respeta las entradas, debemos tener en cuenta:

(I) El tipo de instrucción, es decir, el valor de b, que se obtiene como $l(r(y))$. Una instrucción que **no** respeta las entradas tendrá el valor 1 (correspondiente a $V \leftarrow V + 1$) o el valor 2 (que codifica $V \leftarrow V \dot{-} 1$).

(II) La variable involucrada, es decir, el valor de c, que se obtiene como $r(r(y))$. Una instrucción que **no** respeta las entradas tendrá aquí un número correspondiente a una variable X_i. Como las variables se enumeran

$$Y, X_1, Z_1, X_2, Z_2, X_3, Z_3, \ldots$$

las variables X_i son todas aquellas que ocupan posiciones pares. No obstante, como el número de la variable se disminuye en 1 al codificar la instrucción, las instrucciones que no respetan las entradas tendrán un valor de c impar.

En resumen, la instrucción codificada por y **no** respeta las entradas si y solo si $l(r(y)) \in \{1, 2\}$ y, simultáneamente, $r(r(y))$ es un número impar. Podemos, entonces, reescribir $\tilde{r}_2$ como sigue:

$$\tilde{r}_2(y) = \neg \Big(\big(l(r(y)) = 1 \vee l(r(y)) = 2 \big) \wedge \mathit{impar}\big(r(r(y))\big) \Big)$$

de donde se sigue que $\tilde{r}_2$ es primitivo recursivo, como queríamos demostrar.

4. Programa universal, STP y SNAP

Ejercicio 4

Demostrar que la siguiente función es $\mathcal{S}$-parcial computable.

$$f_1(x, y, z) = \begin{cases} 1 & \text{si la ejecución de } \Phi_x^{(1)}(z) \text{ termina estrictamente en} \\ & \text{menos pasos que la ejecución de } \Phi_y^{(1)}(z) \\ \uparrow & \text{si no} \end{cases}$$

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

Resolución

Construyamos un programa en $\mathcal{S}$ que compute f_1. La idea es utilizar la función $\text{STP}^{(1)}$ para "seguir el rastro" del programa con número x, y así descubrir cuántos pasos tarda en terminar de ejecutar, si es que en algún momento lo hace. Luego, podemos usar nuevamente $\text{STP}^{(1)}$ 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 y solo si esto último resulta ser falso.

Como el predicado $\text{STP}^{(1)}$ es primitivo recursivo, es también $\mathcal{S}$-computable, así que podemos usarlo directamente dentro de nuestros programas (por supuesto, no olvidemos que se trata de una macro que, en rigor, debería ser expandida para obtener un programa válido).

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

$$\begin{array}{ll} [A] & Z_2 \leftarrow \text{STP}^{(1)}(X_3, X_1, Z_1) \\ & \text{IF } Z_2 \neq 0 \text{ GOTO } B \\ & Z_1 \leftarrow Z_1 + 1 \\ & \text{GOTO } A \\ [B] & Z_2 \leftarrow \text{STP}^{(1)}(X_3, X_2, Z_1) \\ & \text{IF } Z_2 \neq 0 \text{ GOTO } B \\ & Y \leftarrow 1 \end{array}$$

Las primeras cuatro instrucciones del programa conforman un ciclo, que se repite hasta que la variable Z_1 contiene exactamente la cantidad de pasos que tarda en terminar el programa con número X_1 cuando recibe la entrada X_3. Si, para esta entrada, el programa X_1 no termina, nuestro programa queda atrapado en este ciclo y se indefine, cumpliendo con el comportamiento esperado para este caso.

Si el ciclo termina, se ejecuta la quinta instrucción, que evalúa la terminación del programa con número X_2 tras la cantidad de pasos que quedó almacenada en Z_1. Si el resultado es 0, entonces X_2 tarda estrictamente más en terminar que X_1 (pudiendo, quizá, no terminar nunca), por lo que devolvemos 1. En caso contrario, es decir, si X_2 termina en una cantidad de pasos menos o igual que X_1, nuestro programa se indefine, quedando atrapado en un ciclo infinito entre las instrucciones quinta y sexta.

También podemos resolver este ejercicio sin escribir explícitamente un programa en $\mathcal{S}$. Para eso, demostraremos previamente un resultado sencillo que nos será útil para este tipo de ejercicios: si un predicado $p : \mathbb{N}^n \to \{0, 1\}$ es $\mathcal{S}$-computable, también es $\mathcal{S}$-parcial computable el *existencial no acotado* sobre este predica-

do, es decir,

$$(\exists t)\ p(x_1, \ldots, x_n, t) = \begin{cases} 1 & \text{si existe } t \text{ tal que } p(x_1, \ldots, x_n, t) = 1 \\ \uparrow & \text{si no} \end{cases}$$

Para probarlo, podemos escribir un sencillo programa en $\mathcal{S}$ que compute dicho predicado. La idea es recorrer todos los posibles valores de t y, para cada uno de ellos, computar p; si existe un valor para el que sea verdadero, en algún momento lo encontraremos y devolveremos 1, mientras que si no existe tal valor, el programa continuará buscándolo de manera indefinida.

$$\begin{array}{ll} [A] & Z_2 \leftarrow p(X_1, \ldots, X_n, Z_1) \\ & \text{IF } Z_2 \neq 0 \text{ GOTO } B \\ & Z_1 \leftarrow Z_1 + 1 \\ & \text{GOTO } A \\ [B] & Y \leftarrow 1 \end{array}$$

Podemos notar que la estructura básica de este programa es, a grandes rasgos, una generalización del que escribimos anteriormente. También, 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 indefinir el existencial, incluso aunque p sea verdadero para otro valor de t que no todavía no hayamos verificado.

Otra manera, incluso más sencilla, de computar el existencial no acotado es reutilizar la *minimización no acotada* vista en las teóricas, es decir, la función

$$\min_t p(x_1, \ldots, x_n, t)$$

que es $\mathcal{S}$-parcial computable siempre que p sea un predicado $\mathcal{S}$-computable. El siguiente programa computa el existencial para un predicado p a partir de esta minimización.

$$\begin{array}{l} Z_1 \leftarrow \min_t p(x_1, \ldots, x_n, t) \\ Y \leftarrow 1 \end{array}$$

Una vez que contamos con el existencial, si encontramos un predicado apropiado, podemos usarlo para reescribir f_1. Al igual que en la solución anterior, intentemos aprovechar el predicado $\mathrm{STP}^{(1)}$, que ya sabemos primitivo recursivo. Las siguientes son dos condiciones necesarias para que $f_1(x, y, z) = 1$:

(I) El programa con número x termina para la entrada z. Es decir, debe existir algún $t \in \mathbb{N}$ tal que $\mathrm{STP}^{(1)}(z, x, t)$ 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 $\mathrm{STP}^{(1)}(z, x, t) \wedge \neg\, \mathrm{STP}^{(1)}(z, y, t)$.

Dado que (II) implica (I), solo tendremos en cuenta la última condición.

Es sencillo advertir que (II) también es condición suficiente 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; es decir, la ejecución de $\Phi_y^{(1)}(z)$ termina en una cantidad de pasos menor o igual que la ejecución de $\Phi_x^{(1)}(z)$. En cualquiera de estas dos situaciones, $f_1(x, y, z) \uparrow$.

Por lo tanto, podemos reescribir

$$f_1(x, y, z) = (\exists t)\big(\, \mathrm{STP}^{(1)}(z, x, t) \wedge \neg\, \mathrm{STP}^{(1)}(z, y, t)\big)$$

donde el predicado al que hace referencia el existencial es primitivo recursivo y, por lo tanto, $\mathcal{S}$-computable (total), de donde se sigue que f_1 es $\mathcal{S}$-parcial computable.

Ejercicio 5

Dada una función $f : \mathbb{N} \to \mathbb{N}$, decimos que $x \in \mathbb{N}$ es un **punto fijo** de f si $f(x) = x$. Demostrar que la siguiente función es $\mathcal{S}$-parcial computable.

$$f_2(x) = \begin{cases} 1 & \text{si } \Phi_x^{(1)} \text{ tiene algún punto fijo} \\ \uparrow & \text{si no} \end{cases}$$

Resolución

De nuevo, empecemos por intentar escribir un programa en $\mathcal{S}$ que compute la función f_2. Una posibilidad es ir verificando el comportamiento del x-ésimo programa para cada una de sus entradas posibles. En el caso de que descubramos que alguna de ellas es un punto fijo, debemos interrumpir la búsqueda y hacer que nuestro programa devuelva 1. De lo contrario, seguiremos buscando indefinidamente y nuestro programa no terminará nunca. El siguiente programa en $\mathcal{S}$ presenta este comportamiento.

$$\begin{array}{ll} [A] & Z_2 \leftarrow \Phi_{X_1}^{(1)}(Z_1) \\ & \text{IF } Z_2 = Z_1 \text{ GOTO } B \\ & Z_1 \leftarrow Z_1 + 1 \\ & \text{GOTO } A \\ [B] & Y \leftarrow 1 \end{array}$$

Ahora bien, analizando con cuidado el código anterior, podemos ver que en realidad no computa correctamente la función f_2. A modo de ejemplo, sea P el siguiente programa:

$$\begin{array}{ll} [A] & \text{IF } X_1 = 0 \text{ GOTO } A \\ & Y \leftarrow 1 \end{array}$$

y sea $e = \#(P)$. Podemos ver fácilmente que

$$\Phi_e^{(1)}(x) = \begin{cases} \uparrow & \text{si } x = 0 \\ 1 & \text{si no} \end{cases}$$

por lo que 1 es un punto fijo de P. Entonces, $f_2(e) = 1$. Sin embargo, si intentamos ejecutar el programa propuesto para f_2, este se indefine en la primera línea, cuando intenta computar $\Phi_e^{(1)}(0)$.

El problema parece ser que, para cada entrada, estamos intentando ejecutar el x-ésimo programa hasta el final; de esta forma, nos arriesgamos a que la ejecución pueda no terminar nunca. Si pudiéramos, en cambio, correr el programa "en paralelo" para todos los valores de entrada posibles, avanzando de a un paso a la vez, podríamos usar $\mathrm{STP}^{(1)}$ para detectar para cuáles de ellos termina y $\mathrm{SNAP}^{(1)}$ para verificar si se trata de puntos fijos, sin quedarnos atrapados a causa de los valores para los que el programa se indefine.

Lamentablemente, esta idea tampoco es factible. Como los valores posibles de entrada son infinitos, verificar para cuáles de ellos el programa termina tras cierta cantidad t de pasos demandaría infinito tiempo. Así, nunca llegaríamos a avanzar más allá de $t = 0$.

Podemos ver que una solución satisfactoria a este problema debería permitirnos recorrer, en algún orden, todas las combinaciones posibles del tipo

(valor de entrada, cantidad de pasos de la ejecución)

sin dejar afuera ninguna de ellas. Ahora bien, estas combinaciones no son más que pares de números naturales. Recorrerlos en forma ordenada equivale a poner estos pares en biyección con $\mathbb{N}$, algo que ya sabemos hacer, gracias a la función codificadora de pares. En otras palabras, como a cada par le corresponde un único número natural, si recorremos primero el par que se codifica con 0, luego el par que se codifica con 1, a continuación el que se codifica con 2, y así sucesivamente, la biyectividad de la codificación nos permite estar seguros de que vamos a recorrerlos todos.

Partiendo de estas ideas, es posible escribir un programa en $\mathcal{S}$ que compute f_2. Sin embargo, en lugar de hacer esto, tratemos de aplicarlas para llevar el problema a la forma de un existencial no acotado. Una manera sencilla de escribir como existencial la propiedad expresada por f_2 es

$$(\exists y)\left(\Phi_x^{(1)}(y)\downarrow \wedge \Phi_x^{(1)}(y) = y\right)$$

Ahora bien, no podemos estar seguros de que este existencial sea $\mathcal{S}$-parcial computable, porque no está construido a partir de un predicado $\mathcal{S}$-computable: la ejecución del programa x, para alguna entrada y, podría indefinirse. Podemos notar que esta formulación es análoga a nuestro primer intento de programa en $\mathcal{S}$.

Podemos empezar tratando de reemplazar las apariciones del intérprete por las funciones $\text{STP}^{(1)}$ y $\text{SNAP}^{(1)}$. Esto, sin embargo, hace aparecer una nueva variable t, que queda ligada a un segundo existencial.

$$(\exists y)\left((\exists t)\left(\text{STP}^{(1)}(y, x, t) \wedge r\big(\text{SNAP}^{(1)}(x, y, t)\big)[1] = y\right)\right)$$

Recordemos que $\text{SNAP}^{(1)}$ devuelve un par de números, cuya parte derecha es una lista que, en su primera posición, contiene el valor de la variable Y. Como estamos pidiendo un valor de t para el que se cumpla $\text{STP}^{(1)}(x, y, t)$, podemos estar seguros de que, para este t, la ejecución del programa terminó, y por lo tanto, el valor de Y corresponde al resultado final de dicha ejecución.

El existencial interno es $\mathcal{S}$-parcial computable, pero no necesariamente es total, por lo que todavía no podemos afirmar nada acerca de la $\mathcal{S}$-computabilidad de esta expresión. Tampoco sirve de nada alterar el orden de los existenciales, obteniendo

$$(\exists t)\left((\exists y)\left(\text{STP}^{(1)}(y, x, t) \wedge r\big(\text{SNAP}^{(1)}(x, y, t)\big)[1] = y\right)\right)$$

lo cual es análogo a nuestra idea anterior de correr el programa para todas las entradas "en paralelo" e ir incrementando la cantidad de pasos ejecutados.

La solución, como ya vimos, pasa por utilizar la codificación de pares para iterar simultáneamente sobre las variables y y t. Esto nos permite reescribir f_2, a partir de la expresión anterior, utilizando un único existencial

$$(\exists \langle y, t\rangle)\left(\text{STP}^{(1)}(y, x, t) \wedge r\big(\text{SNAP}^{(1)}(x, y, t)\big)[1] = y\right)$$

de donde, utilizando las funciones observadoras de pares, podemos reescribir a f_2 en la forma de un existencial no acotado:

$$f_2(x) = (\exists z)\left(\text{STP}^{(1)}\left(l(z), x, r(z)\right) \wedge r\Big(\text{SNAP}^{(1)}\left(x, l(z), r(z)\right)\Big)[1] = l(z)\right)$$

Es sencillo verificar que el predicado al que hace referencia este existencial es primitivo recursivo y, por lo tanto, $\mathcal{S}$-computable (total). Esto nos permite afirmar que f_2 es $\mathcal{S}$-parcial computable, concluyendo la demostración.

Ejercicio adicional

Demostrar que, para todo $n \in \mathbb{N}$, $n \geqslant 1$, la siguiente función es $\mathcal{S}$-parcial computable.

$$f_3(x, y) = \begin{cases} 1 & \text{si existe } \bar{z} \in \mathbb{N}^n \text{ tal que } \Phi_x^{(n)}(\bar{z}) \downarrow, \Phi_y^{(n)}(\bar{z}) \downarrow \text{ y } \Phi_x^{(n)}(\bar{z}) \neq \Phi_y^{(n)}(\bar{z}) \\ \uparrow & \text{si no} \end{cases}$$

Resolución

Haremos la demostración para un $n \geqslant 1$ cualquiera.

Al igual que en los ejercicios anteriores, es posible llevar la propiedad a la que hace referencia f_3 a la forma de un existencial. En este caso, una traducción casi directa del enunciado nos deja con

$$(\exists \bar{z}) \left(\Phi_x^{(n)}(\bar{z}) \downarrow \wedge \Phi_y^{(n)}(\bar{z}) \downarrow \wedge \Phi_x^{(n)}(\bar{z}) \neq \Phi_y^{(n)}(\bar{z}) \right)$$

Reescribiendo la condición del existencial en términos de las funciones $\text{STP}^{(n)}$ y $\text{SNAP}^{(n)}$, obtenemos

$$(\exists \bar{z}) \Big((\exists t) \Big(\text{STP}^{(n)}(\bar{z}, x, t) \wedge \text{STP}^{(n)}(\bar{z}, y, t) \wedge r\big(\text{SNAP}^{(n)}(\bar{z}, x, t)\big)[1] \neq r\big(\text{SNAP}^{(n)}(\bar{z}, y, t)\big)[1] \Big) \Big)$$

El problema de esta expresión es el mismo que se nos presentaba antes: tenemos dos existenciales anidados. A esto se le suma el hecho de que el existencial más externo no predica sobre un número natural, sino sobre un elemento de $\mathbb{N}^n$. En definitiva, en la expresión entran en juego $n + 1$ variables.

La manera de resolverlo es, nuevamente, iterar sobre todas las variables simultáneamente utilizando la codificación de tuplas. En este caso, no podemos usar pares, sino que necesitaremos $n+1$-uplas. Esto no es un problema, dado que, para cualquier $n \geqslant 1$, las $n + 1$-uplas están en biyección con $\mathbb{N}$ y, más aún, las funciones codificadoras y observadoras de $n + 1$-uplas son primitivas recursivas.[2]

A partir de esta idea, podemos reescribir la expresión anterior como un único existencial no acotado de la siguiente manera:

$$(\exists \langle z_1, \ldots, z_n, t \rangle) \Big(\text{STP}^{(n)}(z_1, \ldots, z_n, x, t) \wedge \text{STP}^{(n)}(z_1, \ldots, z_n, y, t) \wedge r\big(\text{SNAP}^{(n)}(z_1, \ldots, z_n, x, t)\big)[1] \neq r\big(\text{SNAP}^{(n)}(z_1, \ldots, z_n, y, t)\big)[1] \Big)$$

de donde, formalizando un poco más, podemos reescribir a f_3 como

$$f_3(x, y) = (\exists w) \Big(\text{STP}^{(n)}\left(\pi_1(w), \ldots, \pi_n(w), x, \pi_{n+1}(w)\right) \wedge \text{STP}^{(n)}\left(\pi_1(w), \ldots, \pi_n(w), y, \pi_{n+1}(w)\right) \wedge r\Big(\text{SNAP}^{(n)}\left(\pi_1(w), \ldots, \pi_n(w), x, \pi_{n+1}(w)\right)\Big)[1] \neq r\Big(\text{SNAP}^{(n)}\left(\pi_1(w), \ldots, \pi_n(w), y, \pi_{n+1}(w)\right)\Big)[1] \Big)$$

donde $\pi_i : \mathbb{N} \to \mathbb{N}$ es la función observadora de la i-ésima componente de una $n + 1$-upla, para todo $i \in \{1, \ldots, n + 1\}$.

Como el existencial no acotado hace referencia a un predicado primitivo recursivo (algo que es fácil de ver, una vez superado lo engorroso de la notación), podemos afirmar que f_3 es una función $\mathcal{S}$-parcial computable, como queríamos probar.

[2] Lo demostramos en el ejercicio 12 de la práctica 1.