Funciones $\mathcal{S}$-computables

Lógica y Computabilidad[*]

2do cuatrimestre 2020

Índice

[*] Basado en apuntes de Franco Frizzo, Edwin Pin, Facundo Ruiz y anteriores docentes

Repaso de conceptos

Repasemos algunos aspectos notacionales a considerar para programar en $\mathcal{S}$:

★ Las variables son de tres tipos:

i. input: $X_1,\ X_2,\ X_3,\ \ldots$

ii. local (temporal): $Z_1,\ Z_2,\ Z_3,\ \ldots$

iii. output: Y

X y Z denotan a las variables X_1 y Z_1. En algunos casos podemos usar la metavariable V (posiblemente indexada) para describir propiedades de programas en general.

★ Las etiquetas se notan como $A,\ B,\ C,\ D,\ E,\ \ldots,\ A_1,\ B_1,\ C_1,\ D_1,\ E_1, \ldots$
Usualmente E indica la finalización de un programa por lo que no suele etiquetar una instrucción. No es este el caso de las variantes indexadas ($E_1,\ E_2,\ \ldots$) que además suelen usarse al expandir una macro. Para describir una etiqueta general usamos la metavariable L.

★ Hay tres tipos de instrucciones básicas (que en un programa pueden estar etiquetadas o no):

i. $V \leftarrow V + 1$

ii. $V \leftarrow V - 1$

iii. IF $V \neq 0$ GOTO L

Una instrucción I con etiqueta L tiene la forma: $[L] \quad I$.

★ Un programa $\mathcal{P}$ es una lista finita de instrucciones de $\mathcal{S}$.

★ Una *macro* es una pseudoinstrucción de $\mathcal{S}$ utilizada para generar subrutinas en programas más grandes y abreviar procesos que se utilicen recurrentemente. Suelen tener una interpretación clara y se fijan por convención. Entre ellas (con su interpretación natural) tenemos:

- Asignación de constantes: $V \leftarrow k$, con $k \in \mathbb{N}$
- Asignación de variables: $V \leftarrow V'$. Aquí distinguimos la instrucción *dummy* $V \leftarrow V$.
- Suma de dos variables: $V \leftarrow V_1 + V_2$. Nótese que V_1 y V_2 podrían ser V.
- Salto incondicional: GOTO L
- Condicional especial: IF $p(\overline{x})$ GOTO L, donde $p(\overline{x})$ es un predicado computable.

Al expandir una macro en un programa $\mathcal{P}$ hay que considerar que:

* sus variables locales no deben coincidir con las que ya aparecen en $\mathcal{P}$ (“variables frescas"),
* sus etiquetas locales no deben coincidir con las que ya aparecen en $\mathcal{P}$ (“etiquetas frescas"),
* para toda instrucción GOTO E_i debe haber al menos una instrucción en $\mathcal{P}$ con etiqueta E_i,
* si la macro está etiquetada en $\mathcal{P}$, la etiqueta se corresponde con su primera instrucción al expandirse.

Partiendo de que en PR sólo se definen funciones totales a partir de las iniciales, en $\mathcal{S}$ podemos construir programas que computen funciones parciales. Por ejemplo, el siguiente programa computa a *la función vacía*:

$$\begin{array}{ll} [A] & \text{IF } Y \neq 0 \text{ GOTO } E \\ & \text{GOTO } A \end{array}$$

$\mathcal{S}$ es esencialmente más poderoso a nivel expresivo que PR. De hecho, es universalmente aceptado que "cualquier planteamiento formal que capture la idea de algoritmo es equivalente al lenguaje $\mathcal{S}$ ". Esta es una de las conclusiones de la tesis de Church-Turing.

Programas y macros

Ejercicio 1: Números que cumplen un predicado

Sea $p(x)$ un predicado computable. Demostrar que la siguiente función es parcial computable:

$$EX_p(r) = \begin{cases} 1 & \text{si hay al menos } r \text{ números } n \text{ tales que } p(n) = 1 \\ \uparrow & \text{si no} \end{cases}$$

Resolución

Una manera de demostrar que una función es parcial computable es construir un programa en $\mathcal{S}$ que la compute. Esto es lo que haremos con EX_p a través de un proceso iterativo sobre cada número natural n, en el que evaluaremos la condición $p(n) = 1$ hasta conseguir r valores. En caso no existir r, el programa se seguirá ejecutando provocando una indefinición.

De esa forma tenemos:

	$Z_1 \leftarrow X$	macro de asignación constante
	$Z_2 \leftarrow 0$	instrucción innecesaria pues Z_i se inicializa en 0
	$Z_3 \leftarrow 0$	instrucción innecesaria
$[B]$	IF $p(Z_2)$ GOTO A	macro de condición especial sobre el predicado p
$[C]$	$Z_2 \leftarrow Z_2 + 1$	
	GOTO B	salto incondicional
$[A]$	$Z_3 \leftarrow Z_3 + 1$	
	IF $Z_1 \leqslant Z_3$ GOTO D	macro de cond. esp. sobre el predicado: $z_1 \leqslant z_3$
	GOTO C	
$[D]$	$Y \leftarrow \alpha(Y)$	macro del operador α

Aquí vemos que usamos la variable Z_2 para iterar sobre $\mathbb{N}$ y Z_3 para acumular la cantidad de valores para los que se satisface p. Además, hacemos uso de una serie de macros para describir este pseudo-programa basadas en operaciones $\mathcal{S}$-computables. Particularmente, podemos demostrar que el predicado $x_1 \leqslant x_2$ lo es:

	$Z_1 \leftarrow X_1$
	$Z_2 \leftarrow X_2$
$[B]$	IF $Z_1 \neq 0$ GOTO C
	$Y \leftarrow \alpha(Y)$
	GOTO E
$[C]$	IF $Z_2 = 0$ GOTO E
	$Z_1 \leftarrow Z_1 - 1$
	$Z_2 \leftarrow Z_2 - 1$
	GOTO B

Ejercicio 2: Multiplicación de dos números

(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:

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

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:

- 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 $\Psi_Q^{(3)}(x, y, z) = x \cdot y$.

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

Cómputos

Ejercicio 3: Cantidad de pasadas por una instrucción

Sea P un programa que contiene exactamente una instrucción (etiquetada o no):

$$Y \leftarrow Y + 1$$

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 una idea de por qué la afirmación es cierta. En principio, sabemos que $\Psi_P^{(1)}(x)$ es el resultado de la ejecución del programa P, que se almacenará en la variable Y al finalizar. Además, esta se inicializa en 0 y sólo puede incrementarse en uno con la instrucción:

$$Y \leftarrow Y + 1.$$

Ahora, como sólo hay una de este tipo en el programa, para que Y llegue a este valor esta necesariamente debe ser ejecutada al menos $\Psi_P^{(1)}(x)$ veces. No obstante, este argumento no es lo suficientemente formal para constituir una demostración. Por ende, necesitamos recurrir a las herramientas de la definición rigurosa de la *semántica* del lenguaje $\mathcal{S}$: estados, descripciones instantáneas y cómputos.

Recordemos que las *descripciones instantáneas* representan completamente la ejecución de un programa en un momento dado. Además, un programa parte de una *inicial* en base al valor de su input y genera un cómputo a través de sus sucesoras hasta llegar a la última de ellas, que contendrá su resultado en la variable Y (o generan una sucesión infinita).

Esto nos da una *estructura inductiva* sobre la que podamos razonar la ejecución de un programa: nos basta que una propiedad valga para la *descripción inicial* y la sucesora de una para la que vale, para probar que lo hace al final de la ejecución. En nuestro caso, "el valor de Y es menor o igual que la cantidad de veces que se pasó por la instrucción de incremento" (o sea que Y no puede superar en valor a esta cantidad). En el esquema inductivo tenemos:

Dado cualquier $x \in \mathbb{N}$ tal que $\Psi_P^{(1)}(x) \downarrow$, donde $d_0, \ldots, d_n$ es el cómputo del programa P a partir de la entrada $x \in \mathbb{N}$ ($d_0 = (1, \sigma_1)$ es la descripción inicial correspondiente). Llamando:

- $\diamondsuit$ m al índice de la instrucción de P que incrementa la variable Y.
- $\diamondsuit$ 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.
- $\diamondsuit$ $d_i[Y]$ al valor de Y indicado por la descripción d_i.

Queremos ver que para todo $i = 0, \ldots, n$, se cumple $k_i \geqslant d_i[Y]$.

(C.B.) Por definición, sabemos que en el estado inicial, $d_0[Y] = 0$ y $k_0 = 0$ (no se ejecutaron instrucciones). Luego, $k_0 \geqslant d_0[Y]$.

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

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

II. Si $j_i \neq m$, bien puede ser que la instrucción no involucre a la variable Y, que lo haga y no la modifique (condicional) o la decremente en 1. En los tres casos, se da que $d_i[Y] \geqslant d_{i+1}[Y]$. Como además $k_{i+1} = k_i$, tenemos que:
$$k_{i+1} = k_i \geqslant d_i[Y] \geqslant d_{i+1}[Y].$$

De allí concluimos que la propiedad se preserva a lo largo de todos los pasos del programa, particularmente al final de su ejecución, que es lo que queríamos probar.

Codificación de programas

Ejercicio 4: Programa con saltos al final

Decimos que un programa P *tiene un salto al final* si posee alguna instrucción de la forma:

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

donde ninguna instrucción 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 de número x consiste en las instrucciones $I_1,\ I_2,\ I_3,\ \ldots,\ I_k$, entonces:

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

Luego, podríamos resolver el problema analizando el código de cada una de estas instrucciones y verificando si alguna contiene un *salto al final*. Más aún, de ver que el predicado

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

sea primitivo recursivo nos permite reescribir a r_1 como

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

Siendo las demás funciones que aparecen en la composición (el existencial acotado, los observadores de listas, etc.) primitivas recursivas, probaríamos que r_1 es primitivo recursivo. Esto nos da una pauta para analizar más detalladamente a $\tilde{r}_1$.

Para esto 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. Consecuentemente, para ver si una instrucción tiene un *salto al final* nos basta con ver el valor de b, que para una instrucción y se obtiene con $l(r(y))$. Además, y es de tipo IF $V \neq 0$ GOTO L si y sólo si $l(r(y)) \geqslant 3$, generando que $\#(L) = l(r(y)) \dot{-} 2$.

También debemos verificar si la etiqueta L en cuestión aparece en alguna instrucción del programa, para lo cual vemos el valor a de la codificación de cada instrucción. El de y puede obtenerse como $l(\tilde{y})$.

En conclusión, podemos reescribir al predicado $\tilde{r}_1$ 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 verifica si *"una instrucción codificada como y es un salto condicional, y no existe ningún t tal que la t-ésima instrucción del programa de número x esté etiquetada por la misma etiqueta a la que apunta el salto de y"*. Notando que el predicado es p.r. concluimos la demostración.

Ejercicio 5: Programa que respeta sus entradas

Decimos que un programa P *respeta sus entradas* si para ningún $i \in \mathbb{N}$ tiene instrucciones de los tipos:

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

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

Vamos a tener una resolución con una estructura general muy similar a la del ejercicio anterior. En este caso, partiendo de la definición de cuándo un programa de $\mathcal{S}$ *respeta sus entradas* definimos el predicado

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

que nos permite centrarnos en el número de cada instrucción. Probando también que $\tilde{r}_2$ es primitivo recursivo, el problema estaría resuelto, ya que podemos reescribir a r_2 como:

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

y de allí ver que es p.r.

En este caso, para ver si una instrucción y codificada como $y = \langle a, \langle b, c\rangle\rangle$ *respeta sus entradas* debemos ver que:

(i) El tipo de instrucción, es decir, el valor de b, que se obtiene por $l(r(y))$, **no** sea 1 (de $V \leftarrow V+1$) ni 2 ($V \leftarrow V \dot{-} 1$).

(ii) La variable involucrada, es decir, el valor de c, que se obtiene por $r(r(y))$, **no** sea de input (X_i). Como las variables se enumeran:

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

todas las X_i ocupan posiciones pares. No obstante, como el número de la variable se disminuye en 1 al codificar la instrucción, debemos ver que su valor no sea impar.

En resumen, la instrucción codificada por y **no** *respeta sus entradas* si y sólo si $l(r(y)) \in \{1, 2\}$ y, simultáneamente, $r(r(y))$ es un número impar. Así, reescribimos a $\tilde{r}_2$ 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 vemos que $\tilde{r}_2$ es primitivo recursivo, como queríamos demostrar.

Programa universal, STP y SNAP

Ejercicio 6: Programa termina antes que otro

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

Necesitamos construir un programa en $\mathcal{S}$ que compute f_1. Para esto vamos a hacer uso de la función $\text{STP}^{(1)}$ que nos permite paso a paso "seguir el rastro" del programa de número x y así detectar cuándo termina de ejecutar (si lo hace). Con el programa y podemos hacer lo mismo y, de terminar, verificar si lo hace en una cantidad menor o igual de pasos que x. Sólo en este caso la salida tendrá valor 1.

Notemos cómo podemos aprovechar que el predicado $\text{STP}^{(1)}$ es primitivo recursivo, y por ende $\mathcal{S}$-computable, para usarlo en nuestro programa a través de una macro (tomando las precauciones necesarias a la hora de expandirla). De esa forma, podemos formar lo siguiente:

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

En las primeras instrucciones formamos un ciclo que va acumulando en la variable Z_1 la cantidad exacta de pasos que tarda en terminar el programa de número X_1 con entrada X_3. En caso no estar definido para dicha entrada el ciclo no terminará, indefiniendo la salida.

De otra forma, con las instrucciones siguientes verificamos si para esta cantidad de pasos el programa en X_2 con entrada X_3 termina. Si no es así, este tarda estrictamente más pasos que X_1 (si es que termina), por lo que retornamos 1. En caso contrario, X_2 termina en una cantidad de pasos menor o igual, por lo que forzamos a que la salida se indefina a través de un ciclo infinito.

Podemos también resolver este ejercicio sin escribir 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, 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 programa simple en $\mathcal{S}$ que compute a este predicado pasando por todos los posibles valores de t y, para cada uno de ellos, computar p; si existe un valor para el

que valga, o podremos encontrar y retornar 1, mientras que si no, el programa lo seguirá buscando indefinidamente:

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

Podemos observar que la estructura básica de este programa es, a grandes rasgos, una generalización de lo que escribimos anteriormente. Es importante notar que para que sea válido p *debe ser total*. Si no, la ejecución podría quedar atrapada en el cómputo de p para algún valor de t e indefinir el cuantificador aún si existiera otro valor para el que p verifique.

Otra manera, aún más sencilla, de computar el existencial no acotado es reutilizar la definición de *minimización no acotada* vista en las teóricas: $\min\limits_t\ p(x_1,\ \ldots,\ x_n,\ t)$ 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 dicha minimización.

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

En base a esto, podemos reescribir f_1 usando el predicado apropiado, aprovechando, como antes, que el predicado $\mathrm{STP}^{(1)}$ es primitivo recursivo. Para ello vemos que para que $f_1(x,\ y,\ z) = 1$ es necesario que:

(I) El programa de número x termine para la entrada z. Es decir, debe existir algún $t \in \mathbb{N}$ para el que $\mathrm{STP}^{(1)}(z,\ x,\ t)$ sea verdadero.

(II) El programa de número y, para la entrada z, *no* termine antes ni al mismo tiempo que el programa x para la misma entrada. Equivalentemente, debe existir un paso t para el que el programa x ya haya terminado, pero y todavía no; o, más formalmente, existe algún $t \in \mathbb{N}$ tal que $\mathrm{STP}^{(1)}(z,\ x,\ t) \wedge \neg\,\mathrm{STP}^{(1)}(z,\ y,\ t)$.

Dado que (II) implica (I), sólo tomaremos en cuenta la última condición. Esto podemos verlo siendo que (II) es condición suficiente para que $f(x,\ y,\ z) = 1$. De no cumplirse, o bien $\Phi_x^{(1)}(z)$ no termina o, para todo t para el que la ejecución de $\Phi_x^{(1)}(z)$ termina en t o menos pasos, la ejecución de $\Phi_y^{(1)}(z)$ también lo hace. En ambas situaciones, $f_1(x,\ y,\ z)\uparrow$.

Por lo tanto, podemos reescribir a f_1 como

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

donde, como el predicado del existencial es primitivo recursivo, también es $\mathcal{S}$-computable (total), y luego f_1 es $\mathcal{S}$-parcial computable.

Ejercicio 7: Punto fijo

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

Nuevamente, empezaremos intentando escribir un programa en $\mathcal{S}$ que compute a la función f_2. Una posibilidad es ir verificando el comportamiento del x-ésimo programa para cada una de sus entradas posibles. En caso de descubrir que alguna es un *punto fijo* interrumpimos la búsqueda y hacemos que nuestro programa devuelva 1. De lo contrario, seguimos buscando indefinidamente y nuestro programa no terminará nunca. Con ello llegamos a lo siguiente:

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

Ahora bien, analizando más cuidadosamente el código anterior, podemos ver que este no computa realmente a f_2. A modo de ejemplo, tomamos el siguiente programa P:

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

y siendo $e = \#(P)$ podemos ver fácilmente que

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

por lo que 1 es un punto fijo de P. Entonces, $f_2(e) = 1$. Sin embargo, al intentar ejecutar el programa propuesto para f_2, se indefinirá al intentar computar $\Phi^{(1)}_e(0)$.

Parece ser entonces que el problema radica en intentar ejecutar el x-ésimo programa hasta el final para toda entrada; arriesgándonos a que no termine nunca. Si en cambio pudiéramos ejecutarlo “en paralelo" para todos los valores de entrada posibles, avanzando de a un paso a la vez, podríamos utilizar $\text{STP}^{(1)}$ para detectar si para alguno termina y de ser así, con $\text{SNAP}^{(1)}$ si se trata de un punto fijo. Así evitamos quedar atrapados entradas por fuera del dominio del programa.

Desafortunadamente no es factible hacerlo considerando que la cantidad de valores es infinita. Es decir, verificar para una cantidad t de pasos generaría que nunca lleguemos a avanzar más allá de $t = 0$. Siendo así, una solución satisfactoria debería permitirnos recorrer, en algún orden, todas las combinaciones posibles del tipo

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

sin dejar afuera a ninguna de ellas.

Podemos observar que estas combinaciones no son más que pares de números naturales y recorrerlos en forma ordenada equivale a ponerlos en biyección con $\mathbb{N}$, lo cual sabemos que podemos hacer con la función codificadora de pares. Dicho de otra manera, como a cada par le corresponde un único número natural, de recorrerlos en el orden en que se codifican (primero el que se codifica con 0, luego 1 y así sucesivamente) nos aseguramos de recorrerlos todos.

De esta forma podemos escribir un programa en $\mathcal{S}$ que compute a f_2. Sin embargo, veremos si podemos trasladar el problema a la forma de un existencial no acotado. Una manera sencilla de hacerlo es tomar el predicado de su definición y escribir

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

Pero recordemos que no podemos estar seguros de que este existencial sea $\mathcal{S}$-parcial computable, y que no está construido a partir de un predicado $\mathcal{S}$-computable: la ejecución del programa x, para

alguna entrada y, podría indefinirse. Este caso es análogo a nuestra primera formulación del programa en $\mathcal{S}$.

Intentamos entonces reemplazar las apariciones del intérprete por las funciones $\text{STP}^{(1)}$ y $\text{SNAP}^{(1)}$, lo cual genera que aparezca una nueva variable t, ligada a un segundo existencial.

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

Recordemos que $\text{SNAP}^{(1)}$ devuelve una descripción instantánea, que en su derecha contiene una lista que comienza con el valor de la variable Y. Como buscamos un valor de t para el que se cumpla $\text{STP}^{(1)}(x,\ y,\ t)$, vemos que para este la ejecución terminó y el valor de Y se corresponde con su salida.

Ahora, viendo que el existencial interno es $\mathcal{S}$-parcial computable, pero no necesariamente es total, no podemos afirmar que la expresión sea computable. El resultado no se ve alterado al invertir los existenciales, debido a que tenemos

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

que podemos analogizar con nuestra idea anterior de recorrer todas las entradas “en paralelo".

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

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

Este es un abuso de notación ya que deberíamos utilizar las funciones observadoras de pares y llegar al siguiente existencial no acotado:

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

Es sencillo verificar que el predicado referenciado por este existencial es primitivo recursivo y, por lo tanto, $\mathcal{S}$-computable (total). Finalmente, podemos afirmar que f_2 es $\mathcal{S}$-parcial computable, concluyendo la demostración.

Ejercicio 8: Salida distinta con misma entrada

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.

Como en los casos anteriores, podemos llevar la propiedad referencia por f_3 a la forma de un existencial. En este caso, una traducción casi directa del enunciado nos deja con

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

y reescribiendo la condición del existencial con las funciones $\mathrm{STP}^{(n)}$ y $\mathrm{SNAP}^{(n)}$, obtenemos

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

Aqui vemos que tenemos el mismo problema que antes: dos existenciales anidados. Sumado a esto, el 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.

Para resolver esto debemos iterar sobre todas las variables simultáneamente utilizando la codificación de tuplas. En este caso no podemos usar pares, sino que necesitaremos $n+1$-uplas, lo cual no es un problema ya 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 p.r.[2]

Con esta idea, podemos reescribir la expresión anterior usando un único existencial no acotado de la siguiente manera:

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

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

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

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 (que podemos ver más allá de la notación engorrosa), podemos afirmar que f_3 es una función $\mathcal{S}$-parcial computable, como queríamos probar.

[2] Se puede demostrar en el ejercicio 12 de la práctica 1.