Lógica y Computabilidad

Gianfranco Zambonni

5 de enero de 2020

Índice

I Computabilidad **4**

1. Máquina de Turing **4**
1.1. Tabla de instrucciones 4
1.2. Definición mátematica 4
1.3. Representación de números y tuplas 5
1.3.1. Números naturales 5
1.3.2. Tuplas 5
1.4. Funciones parciales 5
1.4.1. Cómputo de funciones parciales en máquinas de Turing 6
1.4.2. Poder de cómputo 6

2. Funciones primitivas recursivas y clases PRC **7**
2.1. Funciones iniciales 7
2.2. Clases Primitivas Recursivas Cerradas (PRC) 7
2.3. Funciones primitivas recursivas básicas 8
2.3.1. Predicados primitivos recursivos 9
2.3.2. Operadores lógicos 9
2.3.3. Definición por casos 9
2.3.4. Sumatorias, productorias 10
2.3.5. Cuantificadores acotados 11
2.3.6. Minimización acotada 12
2.3.7. Codificación de tuplas 13
2.3.8. Codificación de secuencias 13

3. Funciones $\mathcal{S}$-Computables **15**
3.1. Sintaxis del lenguaje $\mathcal{S}$ . . . 15
3.1.1. Estado . . . 16
3.1.2. Descripción instantánea . . . 16
3.2. Funciones parciales computables . . . 17
3.2.1. Cómputo . . . 17
3.2.2. Función parcial computable . . . 17
3.2.3. Minimización no acotada . . . 18
3.2.4. Clausura por composición y recursión primitiva . . . 18
3.3. Codificación de programas en $\mathcal{S}$ . . . 19
3.4. Teorema de Cantor . . . 20
3.5. El problema de la detención (Halting Problem) . . . 20
3.6. Universalidad . . . 22
3.6.1. Step counter . . . 23
3.6.2. Snap . . . 23
3.6.3. Teorema de la forma normal . . . 23
3.6.4. Teorema del parámetro . . . 24
3.6.5. Teorema de la recursión . . . 25
3.6.6. Teorema del punto fijo . . . 26

4. Conjuntos computables enumerables **27**

II Lógica **28**

5. Lógica Proposicional **28**

6. Sistemas deductivos para lógica propisicional **28**

7. Lógica de primer orden **28**

8. Completitud y compacidad para lógica de primer orden **28**

A. Otras funciones primitivas recursivas: **29**
A.1. Division . . . 29
A.2. Divisor . . . 29
A.3. i-ésimo primo . . . 29

B. Otras Macros **29**
B.1. $V \leftarrow 0$ . . . 29
B.2. $V \leftarrow k$ . . . 29
B.3. $x = 0$ . . . 29
B.4. $V \leftarrow f(V_1, ..., V_n)$ . . . 29

B.5. IF $p(V_1, \ldots, V_n)$ GOTO L 30

C. Otras definiciones **30**

C.1. La función de Ackerman 30

C.1.1. Versión de Robinson & Peter 31

Parte I
Computabilidad

1. Máquina de Turing

Una máquina de Turing está compuesta por:

- Una **cinta infinita** dividida en celdas que contienen un símbolo de un alfabeto Σ. En esta materia, Σ siempre contiene al símbolo $*$, que representa el blanco, y nunca contiene a L ni a R, que son las etiquetas usadas para indicar al cabezal hacia que lado debe moverse.
- El **cabezal** lee un símbolo y, dependiendo del estado de la máquina, puede escribir uno nuevo o moverse una posición a la derecha o una a la izquierda. Cuando completa una acción, cambia el estado de la máquina.
- Una **tabla finita de instrucciones** que, dado un estado y el símbolo que lee el cabezal, indica que acción debe ser tomada y cual es el estado al que se tiene que pasar.

1.1. Tabla de instrucciones

Cada instrucción es una tupla $(q, s, a, q') \in Q \times \Sigma \times (\Sigma \cup \{L, R\}) \times Q$ tal que:

- $q, q' \in Q$ son estados de la máquina de turing. q es el estado necesario para ejecutar la instrucción y q' el estado en el que queda la máquina después de ejecutarla.
- $s \in \Sigma$ es el símbolo que se tiene que leer cuando la máquina está en q para que la instrucción sea ejecutada.
- $a \in \Sigma \cup \{L, R\}$ es la acción a realizar. Puede ser escribir un símbolo del álfabeto Σ o mover el cabezal hacia alguno de los lados.

Entonces, la tupla (q, s, a, q') se interpreta como "*Si la máquina está en el estado q leyendo en la cinta el símbolo s, entonces realiza la acción a y pasa al estado q'*".

1.2. Definición mátematica

Una máquina de Turing $\mathcal{M}$ es una tupla (Σ, Q, T, q_0, q_f) donde:

- Σ es un conjunto finito de símbolos ($L, R \in \Sigma$ y $* \in \Sigma$)
- Q es un conjunto finito de **estados**, de los cuales dos son:
  - el **estado inicial** q_0
  - y el **estado final** q_f.
- $T \subseteq Q \times \Sigma \times \Sigma$ es la **tabla de instrucciones**.

Hay dos tipos de máquinas de turing:

- **Deterministicas:** Es cuando no hay dos instrucciones que tengan el mismo estado inicial y necesiten leer el mismo símbolo, es decir:

$$\nexists(q_1, s_1, a_1, q_1'), (q_2, s_2, a_2, q_2') \in T \text{ tal que } q_1 = q_2 \ \wedge \ s_1 = s_2$$

- **No deterministicas:** Cuando no es deterministica, osea que cabe la posibilidad que se ejecuten más de una instrucción en un paso y la máquina este en dos o más estados simultáneamente.

$$\exists(q_1, s_1, a_1, q_1'), (q_2, s_2, a_2, q_2') \in T \text{ tal que } q_1 = q_2 \ \wedge \ s_1 = s_2$$

1.3. Representación de números y tuplas

1.3.1. Números naturales

Sea $\sigma = \{*, 1\}$, representaremos a los números naturales en unario (como si usasemos palitos). La representación $\bar{x}$ de $x \in \mathbb{N}$, consta de $x + 1$ palitos.

$$\bar{x} = \underbrace{1...1}_{x+1}$$

Ejemplo: El 0 es "1", el 1 es "11", el 2 es "111", etc.

1.3.2. Tuplas

Las tuplas $(x_1, \ldots, x_n)$ las representamos como una lista de (representaciones de) x_i separados por un blanco $(*)$.

$$*\bar{x_1} * \bar{x_2} * \cdots * \bar{x_n} *$$

Ejemplo: La tupla (1,2) sería $*11 * 111*$

1.4. Funciones parciales

Sea $f : \mathbb{N}^n \to \mathbb{N}$, f es una función parcial si está definida para algunos (tal vez ninguno; tal vez todos) sus argumentos.

Notación:

- $f(x_1, \ldots, x_n) \downarrow$: f está definida para la tupla $(x_1, \ldots, x_n)$ y, en este caso, $f(x_1, \ldots, x_n)$ es un natural.
- $f(x_1, \ldots, x_n) \uparrow$: f se indefine para la tupla $(x_1, \ldots, x_n)$.

Dominio: Conjunto de argumentos para los que f está definida y se nota $\text{dom}(f)$.

$$\text{dom}(f) = \{(x_1, \ldots, x_n) : f(x_1, \ldots, x_n) \downarrow\}$$

Función Total: f es total si está definida para todos sus posibles argumentos:

$$\text{dom}(f) = \mathbb{N}^n$$

1.4.1. Cómputo de funciones parciales en máquinas de Turing

Una función parcial $f : \mathbb{N}^n \to \mathbb{N}$ es **turing computable** si existe una máquina de Turing deterministica $\mathcal{M} = (\Sigma, Q, T, q_0, q_f)$ con $\Sigma = \{*, 1\}$ tal que cuando empieza en la configuración inicial, vale que:

- Si $f(x_1, \ldots, x_n) \downarrow$ entonces, siguiendo sus instrucciones en T, llega al estado q_f,
- Si $f(x_1, \ldots, x_n) \uparrow$ nunca termina en el estado q_f

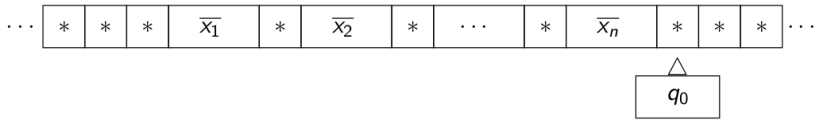

Figura 1: Estado inicial de la máquina de turring

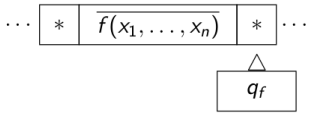

Figura 2: Estado final de la máquina de turing

1.4.2. Poder de cómputo

Sea $f : \mathbb{N}^m \to \mathbb{N}$ una función parcial. Son equivalentes:

1. f es computable en Java
2. f es computable en C
3. f es computable en Haskell
4. f es Turing computable.

2. Funciones primitivas recursivas y clases PRC

2.1. Funciones iniciales

Función calculable de manera efectiva: Son funciones que podemos escribir combinando, de alguna manera, funciones más simples que sabemos que ya sabemos que son efectivas.

La idea es que, dado un **conjunto inicial** de funciones, podamos combinarlas de ciertas formas para conseguir nuevas funciones que permitan realizar cálculos más complejos.

Funciones iniciales:

- $s(x) = x + 1$
- $n(x) = 0$
- **Proyecciones:** $u_i^n(x_1, \ldots, x_n) = x_i$ para $i \in \{1, \ldots, n\}$

Composición: Sea $f : \mathbb{N}^k \to \mathbb{N}$ y $g_1, \ldots, g_k : \mathbb{N}^n \to \mathbb{N}$, entonces $h : \mathbb{N}^n \to \mathbb{N}$ se obtiene a partir de composición de f y $g_1, \ldots, g_k$ por composición si:

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$$

En este contexto, una constante k puede ser definida como:

$$h(t) = s^{(k)}(n(t))$$

Recursión primitiva: $h : \mathbb{N}^{n+1} \to \mathbb{N}$ se obtiene a partir de $g\mathbb{N}^{n+2} \to \mathbb{N}$ y $f : \mathbb{N}^n \to \mathbb{N}$ por recursión primitiva si:

$$h(x_1, \ldots, x_n, 0) = f(x_1, \ldots, x_n)$$
$$h(x_1, \ldots, x_n, t+1) = g(h(x_1, \ldots, x_n, t), x_1, \ldots, x_n, t)$$

2.2. Clases Primitivas Recursivas Cerradas (PRC)

Función primitiva recursiva: Función que se puede obtener a partir de las funciones iniciales por un número finito de aplicaciones de composición y recursión primitiva.

Clase PRC: Una clase $\mathcal{C}$ de funciones totales es **PRC** si

- Las funciones iniciales están en $\mathcal{C}$.
- Si una función f se obtiene a partir de otras pertenecientes a $\mathcal{C}$ por medio de composición o recursión primitiva, entonces f también está en $\mathcal{C}$.

Corolario 2.1. *La clase de funciones primitivas recursivas es una clase PRC.*

Teorema 2.1. *La clase de funciones totales Turing computables es una clase PRC.*

Teorema 2.2. *Una función es p.r. si y solo si pertenece a toda clase PRC.*

DEMOSTRACIÓN

⇐) Si una función f pertenece a todas las clases PRC, en particular, pertenece a la clase de funciones primitiva recursivas. Por lo tanto f es primitiva recursiva.

⇒) Sea f una función primitiva recursiva y sea $\mathcal{C}$ una clase PRC. Como f es p.r, hay una lista $f_1, \ldots, f_n$ tal que

- $f = f_n$ y
- f_i:
  - es una función incial (por lo tanto es p.r y está en $\mathcal{C}$) ó
  - o se obtiene por composición o recursión primitiva a partir de funciones f_j con $j < i$ (luego tambien está en $\mathcal{C}$)

Osea que $f = f_n$ o es una función inicial (y por lo tanto está en $\mathcal{C}$) ó se obtiene por composición o recursión primitiva de algunas de las anteriores (que están en $\mathcal{C}$ y por lo tanto f también lo está). Luego $f \in \mathcal{C}$

□

Corolario 2.2. *La clase de funciones primitivas recursivas es la clase PRC más chica.*

Corolario 2.3. *Toda función p.r. es total y Turing computable*

DEMOSTRACIÓN

Sabemos que la clase de funciones totales Turing computables es PRC. Por el teorema, anterior, si f es p.r, entonces f pertence a todas las clases PRC, en particular a la clase de funciones totales Turing computables. □

2.3. Funciones primitivas recursivas básicas

- Todas las constantes k están en todas las clases PRC.

  $$k = k(t) = s^{(k)}(n(t))$$

- $suma(x, y) = x + y$

  $$\begin{array}{lcl} suma(x, 0) & = & u_1^1(x) \\ suma(x, y+1) & = & g(suma(x, y), x, y) \\ & & \text{donde } g(x_1, x_2, x_3) = s(u_1^3(x_1, x_2, x_3)) \end{array}$$

- $x \cdot y$

$$\begin{aligned} x \cdot 0 &= 0 = n(x) \\ x \cdot (y+1) &= g(x \cdot y, x, y) = x + (x \cdot y) \end{aligned}$$

- x^y

$$\begin{aligned} x^0 &= 1 \\ x^{y+1} &= g(x^y, x, y) = x \cdot x^y \end{aligned}$$

- $factorial(x) = x!$

$$\begin{aligned} 0! &= 1 = s(n(x)) \\ (x+1)! &= g(x!, x) = (x+1) \cdot x! \end{aligned}$$

- $x \dot{-} y = \begin{cases} x - y & \text{si } x \geq y \\ 0 & \text{si no} \end{cases}$

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} (y+1) &= g(x \dot{-} y, x, y) = (x \dot{-} y) \dot{-} 1 \end{aligned}$$

donde $x \dot{-} 1$ es:

$$\begin{aligned} 0 \dot{-} 1 &= 0 \\ (x+1) \dot{-} 1 &= g(x \dot{-} 1, x) = u_2^2(x \dot{-} 1, x) = x \end{aligned}$$

- $\alpha(x) = \begin{cases} 1 & \text{si } x = 0 \\ 0 & \text{si no} \end{cases}$

2.3.1. Predicados primitivos recursivos

Los **predicados** son simplemente funciones que toman valores en $\{0, 1\}$.

- 1 se interpreta como verdadero,
- 0 se interpreta como falso

Los predicados primitivos recursivos son aquellos representados por funciones primitivas recursivas en $\{0, 1\}$.

2.3.2. Operadores lógicos

Teorema 2.3. *Sea $\mathcal{C}$ una clase PRC. Si p y q son predicados en $\mathcal{C}$ entonces $\neg p$, $p \wedge q$ y $p \vee q$ están en $\mathcal{C}$*

DEMOSTRACIÓN

- $\neg p = \alpha(p)$
- $p \wedge q = p \cdot q$
- $p \vee q = \neg(\neg p \wedge \neg q)$

□

Corolario 2.4. *Si p y q son predicados primitivos recursivos también lo son los predicados $\neg p$, $p \vee q$ y $p \wedge q$.*

Corolario 2.5. *Si p y q son predicados totales Turing computables también lo son los predicados $\neg p$, $p \vee q$ y $p \wedge q$*

2.3.3. Definición por casos

Teorema 2.4. *Sea $\mathcal{C}$ una clase PRC. Sean $g_1, \ldots, g_m, h : \mathbb{N}^n \to \mathbb{N}$ funciones en $\mathcal{C}$ y sean $p_1, \ldots, p_m : \mathbb{N}^n \to \{0, 1\}$ predicados mutuamente excluyentes en $\mathcal{C}$. La función:*

$$f(x_1,\ldots,x_n) = \begin{cases} g_1(x_1,\ldots,x_n) & si\ p_1(x_1,\ldots,x_n) \\ \vdots & \\ g_m(x_1,\ldots,x_n) & si\ p_m(x_1,\ldots,x_n) \\ h(x_1,\ldots,x_n) & si\ no \end{cases}$$

está en $\mathcal{C}$

DEMOSTRACIÓN

$$\begin{aligned} f(x_1,\ldots,x_n) =& g_1(x_1,\ldots,x_n)\cdot p_1(x_1,\ldots,x_n) + \ldots \\ &+ g_m(x_1,\ldots,x_n)\cdot p_m(x_1,\ldots,x_n) \\ &+ h(x_1,\ldots,x_n)\cdot(\neg p_1(x_1,\ldots,x_n)\wedge\cdots\wedge\neg p_m(x_1,\ldots,x_n)) \end{aligned}$$

Si algún p_i es verdadero, valdrá 1 y el resto 0 porque todos los predicados son mutuamente excluyentes. Al evaluar f, obtendremos

$$f(x_1,\ldots,x_n) = g_i(x_1,\ldots,x_n) * p_i(x_1,\ldots,x_n) = g_i(x_1,\ldots,x_n)\cdot 1 = g_i(x_1,\ldots,x_n)$$

Si todos los predicados son falsos, entonces todas las g_i se multiplican por cero y, además vale la condición *si no* que es la conjuncion de sus negaciones. Y h es multiplicado por 1. □

2.3.4. Sumatorias, productorias

Teorema 2.5. *Sea* $\mathcal{C}$ *una clase PRC. Si* $f:\mathbb{N}^{n+1}\to\mathbb{N}$ *está en* $\mathcal{C}$ *entonces también están las funciones:*

$$g(y,x_1,\ldots,x_n) = \sum_{t=0}^{y} f(t,x_1,\ldots,x_n))$$
$$h(y,x_1,\ldots,x_n) = \prod_{t=0}^{y} f(t,x_1,\ldots,x_n))$$

DEMOSTRACIÓN

$$\begin{aligned} g(0,x_1,\ldots,x_n) &= f(0,x_1,\ldots,x_n) \\ g(t+1,x_1,\ldots,x_n) &= g(t,x_1,\ldots,x_n) + f(t+1,x_1,\ldots,x_n) \end{aligned}$$

Para h hay que hacer lo mismo pero multiplicando en vez de sumar. □

Teorema 2.6. *Sea $\mathcal{C}$ una clase PRC. Si $f : \mathbb{N}^{n+1} \to \mathbb{N}$ está en $\mathcal{C}$ entonces también están las funciones:*

$$g(y, x_1, \ldots, x_n) = \sum_{t=1}^{y} f(t, x_1, \ldots, x_n))$$

$$h(y, x_1, \ldots, x_n) = \prod_{t=1}^{y} f(t, x_1, \ldots, x_n))$$

La demostración es la misma que la anterios, pero los resultados de los casos bases valen 0 y 1, respectivamente.

2.3.5. Cuantificadores acotados

Sean $p : \mathbb{N}^{n+1} \to \{0, 1\}$ un predicado:

$(\forall t)_{\leq y} p(t, x_1, \ldots, x_n)$ es verdadero si y solo si:

- $p(0, x_1, \ldots, x_n)$ es verdadero **y**

  $\vdots$

- $p(y, x_1, \ldots, x_n)$ es verdadero

$(\exists t)_{\leq y} p(t, x_1, \ldots, x_n)$ es verdadero si y solo si:

- $p(0, x_1, \ldots, x_n)$ es verdadero **o**

  $\vdots$

- $p(y, x_1, \ldots, x_n)$ es verdadero

De la misma manera se pueden definir el existencial y el para todo con $< y$ en vez de $\leq y$.

Teorema 2.7. *Sean $p : \mathbb{N}^{n+1} \to \{0, 1\}$ un predicado perteneciente a una clase PRC $\mathcal{C}$. Entonces los siguientes predicados también están en $\mathcal{C}$:*

$(\forall t)_{\leq y} p(t, x_1, \ldots, x_n)$

$(\exists t)_{\leq y} p(t, x_1, \ldots, x_n)$

DEMOSTRACIÓN

$(\forall t)_{\leq y} p(t, x_1, \ldots, x_n)$ si y solo si $\prod_{t=1}^{y} f(t, x_1, \ldots, x_n)) = 1$

$(\exists t)_{\leq y} p(t, x_1, \ldots, x_n)$ si y solo si $\sum_{t=1}^{y} f(t, x_1, \ldots, x_n)) \neq 0$

Definimos los cuantificadores en función de la sumatoria, productoria y la comparación de igualdad que están en $\mathcal{C}$ (Ver Seccion 2.3.4). Entonces el para todo y el existencial acotados por $\leq$ pertenece a $\mathcal{C}$. □

Teorema 2.8. *Sean $p : \mathbb{N}^{n+1} \to \{0,1\}$ un predicado perteneciente a una clase PRC $\mathcal{C}$. Entonces los siguientes predicados también están en $\mathcal{C}$:*

$(\forall t)_{<y} p(t, x_1, \ldots, x_n)$

$(\exists t)_{<y} p(t, x_1, \ldots, x_n)$

DEMOSTRACIÓN

$(\forall t)_{<y} p(t, x_1, \ldots, x_n)$ si y solo si $(\forall t)_{\leq y}(t = y \vee p(t, x_1, \ldots, x_n)$

$(\exists t)_{<y} p(t, x_1, \ldots, x_n)$ si y solo si $(\forall t)_{\leq y}(t \neq y \wedge p(t, x_1, \ldots, x_n)$

□

2.3.6. Minimización acotada

$$\min_{t \leq y} p(t, x_1, \ldots, x_n) = \begin{cases} \text{mínimo } t \leq y \text{ tal que } p(t, x_1, \ldots, x_n) \text{ es verdadero} & \text{si existe tal } t \\ 0 & \text{si no} \end{cases}$$

Teorema 2.9. *Sean $p : \mathbb{N}^{n+1} \to \{0,1\}$ un predicado de una clase PRC $\mathcal{C}$. Entonces la función $\min_{t \leq y} p(t, x_1, \ldots, x_n)$ también están en $\mathcal{C}$*

DEMOSTRACIÓN

$$\min_{t \leq y} p(t, x_1, \ldots, x_n) = \sum_{u=0}^{y} \prod_{t=0}^{u} \alpha(p(t, x_1, \ldots, x_n))$$

Veamos que esto funciona. Supongamos que t_0 es el mínimo valor que cumple p, entonces:

(1) $p(t_0, x_1, \ldots, x_n) = 1$

(2) $p(t, x_1, \ldots, x_n) = 0$ para todo $t < t_0$

Podemos dividir la sumotaria en dos partes:

$$\underbrace{\sum_{u=0}^{t_0-1} \prod_{t=0}^{u} \alpha(p(t, x_1, \ldots, x_n))}_{(3)} + \underbrace{\sum_{u=t_0}^{y} \prod_{t=0}^{u} \alpha(p(t, x_1, \ldots, x_n))}_{(4)}$$

(3) Son los primeros t_0 términos que corresponden a $u \in [0, t_0 - 1]$. Por (2) sabemos que $p(t, x_1, \ldots, x_n) = 0$ para todo $0 \leq t \leq u < t_0$. Entonces:

$$\sum_{u=0}^{t_0-1} \prod_{t=0}^{u} \alpha(p(t, x_1, \ldots, x_n)) = \sum_{u=0}^{t_0-1} \prod_{t=0}^{u} \alpha(0) = \sum_{u=0}^{t_0-1} \prod_{t=0}^{u} 1 = \sum_{u=0}^{t_0-1} 1 = t_0$$

(4) Resta ver que todos los términos que corresponden a $u \in [t_0, y]$ dan 0 (si diese algo más grande, el resultado final superaría a t_0).

Para esto solo hay que notar que todas la productorias restantes van desde $0 \leq t \leq u$ y que $u \geq t_0$, por lo que en todas se encuentra el valor $\alpha(p(t_0, x_1, \ldots, x_n)) = 0$ (t_0 cumple p). Luego cada una de ellas da cero .

□

2.3.7. Codificación de tuplas

Definimos la función primitiva recursiva

$$\langle x, y \rangle = 2^x(2 \cdot y + 1) - 1$$

Proposición 2.1. *Hay una única solución (x, y) a la ecuación $\langle x, y \rangle = z$.*

- *x es el máximo número tal que $2^x | (z + 1)$*
- $y = (\frac{z+1}{2^x} - 1)/2$

Observadores: Sea $z = \langle x, y \rangle$:

- $l(z) = x$
- $r(z) = y$

Proposición 2.2. *Los observadores de pares son primitivas recursivas*

DEMOSTRACIÓN

Como x e y son menores a $z + 1$ tenemos que:

- $l(z) = \min_{x \leq z} \left((\exists y)_{\leq z} \;\; z = \langle x, y \rangle \right)$
- $r(z) = \min_{y \leq z} \left((\exists x)_{\leq z} \;\; z = \langle x, y \rangle \right)$

2.3.8. Codificación de secuencias

El **número de Gödel** de la secuencia $a_1, \ldots, a_n$ es el número:

$$[a_1, \ldots, a_n] = \prod_{i=1}^{n} p_i^{a_i}$$

donde p_i es el $i-$ésimo primo ($i \geq 1$)

Teorema 2.10. *Si $a_1, \ldots, a_n = b_1, \ldots, b_n$ entonces $a_i = b_i$ para todo $i \in \{1, \ldots, n\}$.*

Observación: La demostración del teorema se basa en que la factorización en primos de cualquier número es única.

Observación: $[a_1, \ldots, a_n] = [a_1, \ldots, a_n, 0] = [a_1, \ldots, a_n, 0, 0] = \ldots$

Observadores: Sea $x = [a_1, \ldots, a_n]$:

- $x[i] = a_i$
- $|x|$ = longitud de x

Proposición 2.3. *Los observadores de secuencias son primitivas recursivas*

DEMOSTRACIÓN

- $x[i] = \min_{t \leq x} \left(\neg \left(p_i^{t+1} | x\right)\right)$

  Es el mínimo t tal que p_i^{t+1} no divide a x. La función p_i que nos devuelve el i-ésimo primo es primitiva recursiva y el predicado de divisibilidad son primitivos recursivos (Ver apéndice A)

- $|x| = \min_{i \leq x} \left(x[i] \neq 0 \wedge (\forall j)_{\leq x} \left(j \leq i \vee x[j] = 0\right)\right)$

3. Funciones $\mathcal{S}$-Computables

3.1. Sintaxis del lenguaje $\mathcal{S}$

En la materia usamos el lenguaje $\mathcal{S}$.

- Usaremos letras como variables cuyos valores serán números enteros no negativos. En particular:
  - Las letras $X_1, X_2, \ldots$ serán las **variables de entradas** de nuestro programa.
  - La letra Y, **variable de salida**. Siempre empieza inicializada en cero.
  - Las letras $Z_1, Z_2, \ldots$, las **variables locales**. Siempre empiezan inicializadas en cero.
- Un programa en $\mathcal{S}$ consistirá de una secuencia fínita de instrucciones. Tendremos de tres tipos:

Instrucción	Interpretación
$V \leftarrow V + 1$	Incrementa en uno el valor de la variable V.
$V \leftarrow V - 1$	Decrementa en uno el valor de la variable V. Si V es cero, no hace nada
IF $V \neq 0$ GOTO L	Si el valor de V no es cero, entonces ejecuta la instrucción con etiqueta L, sino sigue con la siguiente instrucción de la secuencia.

Un programa termina cuando se ejecuta la última instrucción o cuando se realiza un salto condicional a una etiqueta que no pertenece al programa

Macros: Cuando programemos habrá secuencias de intrucciones que usaremos frecuentemente. A éstas, les podremos asignar una abreviación que podremos usar como pseudo instrucciónes.

Cuando utilizemos macros, vamos a asumir que las etiquetas y variables usadas dentro de ellas se remplazan automáticamente por etiquetas y variables que no esté usando nuestro programa

Macro GOTO L : Nos permitirá realizar un salto sin la necesidad de una condición.

GOTO L:
$Z \leftarrow Z + 1$
IF $Z \neq 0$ GOTO L

Programa $Y \leftarrow X$: Asignar a Y el valor de X.

$[A]$ IF $X \neq 0$ GOTO B
GOTO C
$[B]$ $X \leftarrow X - 1$
$Y \leftarrow Y + 1$
$Z \leftarrow Z + 1$A
GOTO A
$[C]$ IF $Z \neq 0$ GOTO D
GOTO E
$[D]$ $Z \leftarrow Z - 1$
$X \leftarrow X + 1$
GOTO C

Si renombramos a X e Y por V_1 y V, respectivamente, conseguimos la macro asociada a este programa.

3.1.1. Estado

Estado de un programa P: Es una lista de equaciones de la forma $V = m$ donde V es una variable y m un número. La lista, incluye una de estas ecuaciones por cada variable que usa (no hay dos ecuaciones para la misma variable).

Programa P:

$[A]$ $X \leftarrow X - 1$
$Y \leftarrow Y + 1$
IF $X \neq 0$ GOTO A

Algunos estados de P:

- $X = 3, Y = 1$
- $X = 3, Y = 1, Z = 0$
- $X = 3, Y = 1, Z = 8$ (si bien no es alcanzable, es un estado válido)

3.1.2. Descripción instantánea

Descripción instantánea (snapshot): Sea P un programa de longitud n. Dado un estado σ de P y un $i \in 1, \ldots, n+1$, el par (i, σ) es una descripción instantanea de P e indica el valor de sus variables antes de ejecutar la i-ésima instrucción.

Descripción terminal: Si $i = n + 1$, entonces la descripción se llama **terminal**.

Para una descripción (i, σ), no terminal podemos definir su **sucesor** (j, τ) de la siguiente manera:

Si la i-ésima instrucción es:

- $V \leftarrow V + 1$ entonces $j = i + 1$ y τ es σ remplazando $V = m$ por $V = m + 1$
- $V \leftarrow V - 1$ entonces $j = i + 1$ y τ es σ remplazando $V = m$ por $V = \text{máx}\{m - 1, 0\}$
- IF $V \neq 0$ GOTO L entonces $\tau = \sigma$ y con j pueden suceder dos cosas:

- Si σ contiene la fórmula $V = 0$ entonces $j = i + 1$
- Si no:
  - Si existe una instrucción en P con etiqueta L, entonces

$$j = \text{mín}\{k : k\text{-ésima instrucción de } P \text{ tiene etiqueta } L\}$$

  - Si no, $j = n + 1$

Notar que podemos tenes más de una instrucción con la misma etiqueta. Sin embargo, nuestra definición de descripción instantanea, interpreta que un condicional siempre se refiere a la primera instrucción que la usa.

3.2. Funciones parciales computables

3.2.1. Cómputo

Estado inicial: Sea P un programa del lenguaje $\mathcal{S}$ y $r_1, \ldots, r_m$ números dados. Formamos el estado inicial σ_1 de P con las ecuaciones $X_1 = r_1,\ X_2 = r_2, \ldots, X_m = r_m, Y = 0$ y $V = 0$ para cada variable V que use el programa y no sean las ya mencionadas. Y $(1, \sigma_1)$ es la **descripción inicial**.

En las definiciones que usaremos, un mismo programa P puede servir para computar funciones de una variable, dos variables, etc. Esto depende de la cantidad de variables de entradas que especifiquemos. Si especificamos menos de las que deberíamos, el resto toma valor 0. Si especificamos más, entonces el programa simplemente las ignorará.

Computo de un programa P: Dado un programa P y una descripción inicial $d_1 = (1, \sigma)$, un cómputo es una secuencia finita de descripciones $d_1, \ldots, d_k$ tal que d_k es una descripción terminal. Si esta secuencia existe, notamos $\Psi_P^{(m)}(r_1, \ldots, r_m)$ al valor de Y en d_k.

En los casos en que la secuencia de instrucciones sea infinita (nunca lleguemos a un estado terminal si partimos desde d_1), diremos que $\Psi_P^{(m)}(r_1, \ldots, r_m)$ está indefinido.

$$\Psi_P^{(m)}(r_1, \ldots, r_m) = \begin{cases} \text{el valor de Y} & \text{si existe un cómputo de } P \text{ desde } (1, \sigma_1) \\ \uparrow & \text{si no} \end{cases}$$

3.2.2. Función parcial computable

Una función (parcial) $f : \mathbb{N}^m \to \mathbb{N}$ es **$\mathcal{S}$-parcial computable** (o **parcial computable**)si existe un programa P tal que

$$f(r_1, \ldots, r_m) = \Psi_P^{(m)}(r_1, \ldots, r_m) \text{ para todo } (r_1, \ldots, r_m) \in \mathbb{N}^m$$

Esto quiere decir, que para $(r_1, \ldots, r_m)$, ambos lados están definidos y tienen el mismo valor o ambos lados están indefinidos.

Función $\mathcal{S}$-computable: (o **computable**) si es parcial computable y total.

3.2.3. Minimización no acotada

Definimos la minimización no acotada como:

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

Teorema 3.1. *Si $p : \mathbb{N}^{n+1} \to \{0, 1\}$ es un predicado computable entonces $\min_t p(t, x_1, \ldots, x_n)$ es parcial computable*

DEMOSTRACIÓN

Si mostramos un programa que computa $\min_t p(t, x_1, \ldots, x_n)$, entonces queda probado el teorema. Ese programa es:

$$\begin{array}{ll} [A] & {}^*\text{IF } p(Y, X_1, \ldots, X_n) \text{ GOTO } E \\ & Y \leftarrow Y + 1 \\ & \text{GOTO } A \end{array}$$

*Ver definición en el apéndice B.5

3.2.4. Clausura por composición y recursión primitiva

Teorema 3.2. *Si $h : \mathbb{N}^n \to \mathbb{N}$ se obtiene a partir de las funciones (parciales) computables $f, g_1, \ldots, g_k$ por composición, osea tiene la siguiente forma:*

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$$

entonces h es (parcial) computable.

DEMOSTRACIÓN

Se puede ver facilmente que el programa mostrado a continuación computa h. En el apéndice B.4, se define la macro que nos permite asignar el resultado de una función a una variable.

$$\begin{array}{c} Z_1 \leftarrow g_1(X_1, \ldots, X_n) \\ \vdots \\ Z_k \leftarrow g_k(X_1, \ldots, X_n) \\ Y \leftarrow f(Z_1, \ldots, Z_k) \end{array}$$

Proposición 3.1. *Si $f, g_1, \ldots, g_k$ son totales entonces h es total.*

Teorema 3.3. *Si h se obtiene a partir de g por recursión primitiva y g es computable entonces h es computable.*

DEMOSTRACIÓN

El siguiente programa computa h. La macro que asigna una constante y el programa que calcula X = 0 se pueden ver en el apéndice B.

$$\begin{array}{ll} & Y \leftarrow f(X_1, \ldots, X_n) \\ [A] & \text{IF } X_{n+1} = 0 \text{ GOTO } E \\ & Y \leftarrow g(Z, Y, X_1, \ldots, X_n) \\ & Z \leftarrow Z + 1 \\ & X_{n+1} \leftarrow X_{n+1} - 1 \\ & \text{GOTO } A \end{array}$$

Proposición 3.2. *Si g es total entonces h es total.*

Teorema 3.4. *La clase de funciones computables es una clase PRC*

DEMOSTRACIÓN

Por los teorema 3.2 y 3.3, la clase de funciones computables está cerrada por composición y recursión primitiva. Osea que solo nos falta ver que las funciones iniciales son computables:

- $s(x) = x + 1$ se computa con el programa:

$$\begin{array}{l} Y \leftarrow X \\ Y \leftarrow Y + 1 \end{array}$$

- $n(x) = 0$ se computa con el programa vacio.
- $u_i^n(x_1, \ldots, x_n)$ se computa con el programa:

$$Y \leftarrow X_i$$

Corolario 3.1. *Toda función primitiva recursiva es computable.*

3.3. Codificación de programas en $\mathcal{S}$

Vamos a asociar cada programa P el lenguaje S con un número $\#(P)$ de tal manera que podamos recuparlo solo conociendo su número. Para esto:

- Agregamos la instrucción $V \leftarrow V$ a $\mathcal{S}$ que no hace nada.
- Le damos un órden a las variables: $Y\ X_1\ Z_1\ X_2\ Z_2\ X_3\ Z_3 \ldots$
- Y hacemos los mismo para las etiquetas: $A_1\ B_1\ C_1\ D_1\ E_1\ A_2\ B_2\ C_2\ D_2\ E_2\ A_3$

Escribimos $\#(V)$ y $\#(L)$ para indicar la posición en la lista de una variable o etiquetas dada. Por ejemplo: $\#(X_2) = 4$, $\#(E_1) = 5$

Ahora, sea I una instrucción etiquetada (o no) del lenguaje S, entonces la podemos representar con una tupla $\#(I) = \langle a, \langle b, c \rangle \rangle$ donde:

1. Si I tiene etiqueta L, entonces $a = \#(L)$. Si no $a = 0$.
2. Si I es una instrucción del tipo:
   - $V \leftarrow V$ entonces $b = 0$
   - $V \leftarrow V + 1$ entonces $b = 1$
   - $V \leftarrow V - 1$ entonces $b = 2$
   - IF $V \neq 0$ GOTO L entonces $b = \#(L) + 2$
3. $c = \#(V) - 1$ siendo V la variable usada en esa instrucción.

Finalmente, sea P un programa que consiste de las instrucciones $I_1, \ldots, I_k$, lo codificamos como una secuencia de instrucciones:

$$\#(P) = [\#(I_1), \ldots, \#(I_k)] - 1$$

Recordemos que tanto las tuplas como las listas pueden ser representadas como números naturales y sus observadores son primitivos recursivos (Ver Secciones 2.3.7 y 2.3.8).

Observemos que, por como habíamos definido las secuencias, pasaba que $[a_1, \ldots, a_k] = [a_1, \ldots, a_k, 0] = [a_1, \ldots, a_k, 0, 0] = [a_1, \ldots, a_k, 0, 0, \ldots]$. Ahora, $0 = \langle 0, \langle 0, 0\rangle\rangle = Y \leftarrow Y$ por lo que cada 0 agrega esta instrucción a un programa.

Si bien los programas $[a_1, \ldots, a_k]$, $[a_1, \ldots, a_k, 0]$ y $[a_1, \ldots, a_k, 0, 0, \ldots]$ son equivalentes, removemos esta ambigüedad pidiendo que ningún programa $\mathcal{S}$ puede terminar con la instrucción $Y \leftarrow Y$. Con esto, cada número representa a un único programa.

3.4. Teorema de Cantor

Teorema 3.5. *El conjunto de las funciones (totales)* $\mathbb{N} \to \mathbb{N}$ *no es numerable.*

DEMOSTRACIÓN

Supongamos que el conjunto mencionado es numerable, entonces puedo enumerar todas las funciones de la siguiente forma: $f_0, f_1, f_2, \ldots$.
Sea $g : \mathbb{N} \to \mathbb{N}$ tal que $g(x) = f_x(x) + 1$. Como g es una función de naturales en naturales, debería estar en la enumeración de funciones que hicimos más arriba. Por lo que $g = f_k$ para algún k.
Sin embargo, $g(k) = f_k(k) + 1 \neq f_k(k)$ para todo k. Luego, g no puede ser igual a ninguna de las funciones enumeradas. Absurdo ($f_0, f_1, \ldots$ era una enumeración de **todas** las funciones $\mathbb{N} \to \mathbb{N}$)

3.5. El problema de la detención (Halting Problem)

Dado un programa P tal que $\#(P) = y$, entonces HALT(x,y) es verdadero si $\Psi_P^{(1)}(x)$ está definido (osea si el programa termina y devuelve un resultado) y falso si $\Psi_P^{(1)}(x)$ está indefinido.

$$\text{HALT}(x,y) = \begin{cases} 1 & \text{si } \Psi_P^{(1)}(x)\downarrow \\ 0 & \text{si no} \end{cases}$$

donde P es el programa tal que $\#(P) = y$

Teorema 3.6. *HALT(x,y) no es un predicado computable.*

DEMOSTRACIÓN

Supongamos que HALT es computable, entonces podríamos construir el siguiente programa P:

$$[A] \quad \text{IF HALT}(X,X) \text{ GOTO } A$$

Es claro, que el cómputo de P es:

$$\Psi_P^{(1)}(x) = \begin{cases} \uparrow & \text{si HALT}(x,x) \\ 0 & \text{si } \neg \text{ HALT}(x,x) \end{cases}$$

Supongamos, ahora, que $\#(P) = e$. Entonces, para algún x, usando la definición de HALT nos queda que:

$$\text{HALT}(x,e) \iff \Psi_P^{(1)}(x)\downarrow \iff \Psi_P^{(1)}(x) = 0 \iff \neg\text{HALT}(x,x)$$

Si hacemos que $x = e$, entonces tenemos que $\text{HALT}(e,e) \iff \neg\text{HALT}(e,e)$. Pero esto es una contradicción. □

Con esto podemos concluir que no existe un algoritmo que nos permita computar $\text{HALT}(x,y)$.

Tesis de Church: Todos las funciones computables sobre naturales, se pueden programar en $\mathcal{S}$.

Diagonalización Dado un conjunto de funciones, la idea es generar una función que debería pertenecer al conjunto pero es distintas a todas las funciones de ese conjunto y, por lo tanto, llegar a una contradicción.

En el teorema de cantor (Teorema 3.5), usamos este método para concluir que había mas funciónes $\mathbb{N} \to \mathbb{N}$ que números naturales. Para esto, dimos un orden a la clase de funciones de este tipo y ordenamos sus valores de la siguiente forma:

$$\begin{array}{cccc} f_0(0) & f_0(1) & f_0(0) & \cdots \\ f_1(0) & f_1(1) & f_1(0) & \cdots \\ f_2(0) & f_2(1) & f_2(0) & \cdots \end{array}$$

Cuando definimos la nueva función que debería pertenecer a la enumeración propuesta, lo hacemos de tal manera que use los valores de la diagonal de esta matriz y sea distinto a todos ellos ($g(x) = f_x(x) + 1$).

Como $g(x)$ usa todas las funciones enumeradas y es distintas a todas ellas, se llega a un absurdo.

3.6. Universalidad

Definimos la función $\Phi^{(n)}(x_1, \ldots, x_n, e)$ como la salida del programa e con entrada $x_1, \ldots, x_n$, es decir $\Phi^{(n)}(x_1, \ldots, x_n, e) = \Psi_P^{(n)}(x_1, \ldots, x_n)$ con P tal que $\#(P) = e$.

Teorema 3.7. *Para cada $n > 0$ la función $\Phi^{(n)}$ es parcial computable.*

DEMOSTRACIÓN

Para demostrar esto vamos a construir el programa U_n que computa $\Phi^{(n)}$.

	$Z \leftarrow X_{n+1} + 1$	Z es la lista de instucciones del programa.
	$S \leftarrow \prod_{i=1}^{n} (p_{2i})^{x_i}$	S es la lista $[0, X_1, 0, X_2, 0, \ldots]$ que codifica su estado inicial (según el orden de variables que habíamos definido en la sección 3.3).
	$K \leftarrow 1$	K es la próxima instrucción a ejecutar.
$[C]$	IF $K = \lvert Z\rvert + 1 \vee K = 0$ GOTO F	El IF se fija si todavía hay instrucciones para ejecutar. Si no las hay, salta a la instrucción con etiqueta F. Si hay, se sigue en la siguiente instrucción.
	$U \leftarrow r(Z[K])$	U guarda el tipo y la variable de la K-ésima instrucción.
	$P \leftarrow p_{r(U)+1}$	P es el primo asociado a la variable de esta instrucción.
	IF $l(U) = 0$ GOTO N	Si el tipo $l(U)$ de la instrucción es $V \leftarrow V$, entonces vamos a N. Si es una suma vamos a S.
	IF $l(U) = 1$ GOTO S	
	IF $\neg(P \mid S)$ GOTO N	La condición $\neg(P \mid S)$ comprueba si la variable mencionada por U está en cero. Si lo está, vamos a N, sino nos fijamos que la instrucción sea, efectivamente, una resta. Si lo es, vamos a R, sino seguimos en la próxima instrucción.
	IF $l(U) = 2$ GOTO R	
	$K \leftarrow \min_{i \leq \lvert Z\rvert} (l(Z[i]) + 2 = l(U))$	
	GOTO C	
$[R]$	$S \leftarrow S$ div P	
	GOTO N	
$[S]$	$S \leftarrow S \cdot P$	
	GOTO N	
$[N]$	$K \leftarrow K + 1$	
	GOTO C	
$[F]$	$Y \leftarrow S[1]$	

Si llegamos a la línea del mínimo, quiere decir que la instrucción era un salto condicional y, además, la variable era distinta de cero por lo quu hay que realizar el salto condicional. Aquí se computa la posición de la instrucción que se debe ejecutar a continuación. Y luego se vuelve a comenzar el ciclo.
En las etiqueta R y S se produce la resta y suma, respectivamente, de la variable correspondiente y luego se salta a N.
En N se incrementa K y se vuelve a comenzar el ciclo.
Finalmente, en F se asigna a Y el valor $S[1]$. □

A veces notaremos

$$\Phi_e^{(n)}(x_1, \ldots, x_n) = \Phi^{(n)}(x_1, \ldots, x_n, e)$$

. Y omitiremos el supra-índice cuando $n = 1$

$$\Phi_e(x) = \Phi_e^{(1)}(x, e)$$

3.6.1. Step counter

Definimos el predicado STP de la siguiente manera:

$\text{STP}^{(n)}(x_1, \ldots, x_n, e, t) \iff$ el programa e con entrada termina en t o menos pasos con la entrada $x_1, \ldots, x_n \iff$ hay un cómputo del programa e de longitud $\leq t + 1$, comenzando con la entrada $x_1, \ldots, x_n$

Teorema 3.8. *Para cada $n > 0$, el predicado $STP^{(n)}(x_1, \ldots, x_n, e, t)$ es primitivo recursivo.*

3.6.2. Snap

La función SNAP no devuelve la configuración instantánea del programa e con entrada $x_1, \ldots, x_n$ en el paso t.

$\text{SNAP}^{(n)}(x_1, \ldots, x_n, e, t) = \langle$ número de instrucción, lista representando el estado $\rangle$

Teorema 3.9. *Para cada $n > 0$, el predicado $SNAP^{(n)}(x_1, \ldots, x_n, e, t)$ es primitiva recursiva.*

3.6.3. Teorema de la forma normal

Teorema 3.10. *Sea $f : \mathbb{N}^N \to \mathbb{N}$ una función parcial computable. Entonces existe un predicado primitivo recursivo $R : \mathbb{N}^{n+1} \to \mathbb{N}$ tal que $f(x_1, \ldots, x_n) = l(\text{mín}_z\, R(x_1, \ldots, x_n, z))$.*

DEMOSTRACIÓN

Sea e el número de programa que computa $f(x_1, \ldots, x_n)$. Vamos a probar que es verdadero usando el predicado:

$$R(x_1, \ldots, x_n, z) = \text{STP}^{(n)}(x_1, \ldots, x_n, e, r(z)) \wedge (l(z) = r(\text{SNAP}^{(n)}(x_1, \ldots, x_n, e, r(z)))[1]$$

Primero, supongamos que $f(x_1, \ldots, x_n) \downarrow$. Entonces $\Phi_e^{(n)}(x_1, \ldots, x_n)$ termina en una cantidad $r(z)$ fínita de pasos. y $\text{STP}^{(n)}(x_1, \ldots, x_n, e, r(z))$ es verdadero. Además, $r(\text{SNAP}^{(n)}(x_1, \ldots, x_n, e, r(z)))$ representa el estado final del programa por lo que tiene almacenado el valor final de la variable Y en su primer posición.
De esta forma, caracterizamos z como la tupla que tiene, a la izquierda, el valor $f(x_1, \ldots, x_n)$ y, a la derecha, la cantidad de pasos que hizo el programa e para computar f con esa entrada.
Por otro lado, si $f(x_1, \ldots, x_n) \uparrow$, entonces $\text{STP}^{(n)}(x_1, \ldots, x_n, e, r(z))$ nunca es verdadero por lo que la función mín se indefine (ya que nunca encuentra un $r(z)$ válido).

A partir del teorema anterior, se derivan los siguientes:

Teorema 3.11. *Una función es **parcial computable** si se puede obtener a partir de las funciones iniciales por un número finito de aplicaciones de composición, recursión primitiva y **minimización**.*

Teorema 3.12. *Una función es **computable** si se puede obtener a partir de las funciones iniciales por un número finito de aplicaciones de composición, recursión primitiva y **minimización propia** (es decir, siempre existe al menos un valor t que hace verdadera la propiedad que se busca minimizar).*

3.6.4. Teorema del parámetro

Teorema 3.13. *Para cada $n, m > 0$, hay una función primitiva recursiva inyectiva $S_n^m : \mathbb{N}^{n+1} \to \mathbb{N}$ tal que:*

$$\Phi_y^{(n+m)}(x_1, \ldots, x_m, u_1, \ldots, u_n) = \Phi^{(m)}_{S_m^n(u_1,\ldots,u_n,y)}(x_1, \ldots, x_m)$$

Supongamos que dado un número de un programa con número y fijamos las variables $u_1, \ldots, u_n$. El teorema del parámetro nos dice que existe un programa con número $S_m^n(u_1, \ldots, u_n, y)(x_1, \ldots, x_m)$ que computa la función que toma como parámetros las m variables restantes. Y, además, ese número se puede obtener de manera computable.

DEMOSTRACIÓN

Se hace por inducción, en la cantidad de parámetros fijados.

Caso $n = 1$: Necesitamos mostrar que $S_m^1(u, y)$ es una función primitiva recursiva tal que $\Phi^{(m+1)}(x_1, \ldots, x_m, u, y) = \Phi^{(m)}(x_1, \ldots, x_m, S_m^1(u, y))$
Sea P el programa tal que $\#(P) = y$, entonces el programa Q tal que $\#(Q) = S_m^1(u, y)$ puede ser el program que primero asigna a X_{m+1} el valor u y luego ejecuta P.
En el ápendice B.2 se ve como asignar un valor cualquiera a una variable ejecuntando la instrucción $X_{m+1} \leftarrow X_{m+1} + 1$ la cantidad necesaria de veces.
El número de esta instrucción es: $\langle 0, \langle 1, 2m + 1\rangle\rangle = 16m + 10$. Por lo que hay que agregar u este número a la lista de instrucciones P, entonces el número de Q se podría computar de la siguiente forma:

$$S_m^1(u, y) = \left[\left(\prod_{i=1}^{u} p_i\right)^{16m+10} \prod_{j=1}^{|y+1|} p_{u+j}^{(y+1)_j}\right] \dot{-} 1$$

Lo que hace es asignar a los primeros u primos, la instrucción mencionada y mover a $p_{u+1}, p_{u+2}, \ldots$ las instrucciones de P. Además, S_m^1 es primitiva recursiva (es nuestra codificación de listas).

Caso inductivo (n = k + 1): Asumimos que el resultado vale para $n = k$. Primero aplicamos el mismo resultado que para el caso base y obtenemos que:

$$\Phi_y^{(m+k+1)}(x_1,\ldots,x_m,u_1,\ldots,u_{k+1}) = \Phi^{(m+k)}_{S^1_{m+k}(u_{k+1},y)}(x_1,\ldots,x_m,u_1,\ldots,u_k)$$

Nuestra hipotesis inductiva es que para cualquier programa existe una función primitiva recursiva S_m^k que nos permite fijar los últimos k parámetros de la entrada. En particular, existe para el programa $S^1_{m+k}(u_{k+1},y)$, entonces:

$$\Phi^{(m+k)}_{S^1_{m+k}(u_{k+1},y)}(x_1,\ldots,x_m,u_1,\ldots,u_k) = \Phi^{(m+k)}_{S^k_m(u_1,\ldots,u_k,S^1_{m+k}(u_{k+1},y))}(x_1,\ldots,x_m)$$

Luego podemos tomar la función primitiva recursiva

$$S_m^{k+1}(u_1,\ldots,u_{k+1},y) = S_m^k(u_1,\ldots,u_k,S^1_{m+k}(u_{k+1},y))$$

□

3.6.5. Teorema de la recursión

En la demostración del Halting Problem (Teorema 3.6) construimos un programa P que, cuando se ejecuta con su mismo número de programa, evidencia una contradicción. El fénomeno de un programa actuando sobre su propia descripción se conoce como *auto-referencia*. El teorema de la recursión nos presenta una técnica para obtener programas *auto-referentes*.

Teorema 3.14. *Si $g:\mathbb{N}^{n+1}\to\mathbb{N}$ es parcial computable, existe un e tal que $\Phi_e^{(n)} = g(e,x_1,\ldots,x_n)$*

DEMOSTRACIÓN

Sea S_n^1 la función del Teorema del parámetro:

$$\Phi_y^{(n+1)}(x_1,\ldots,x_n,u) = \phi^{(n)}_{S^1_n(u,y)}(x_1,\ldots,x_n)$$

Consideremos la función parcial computable

$$h(x_1,\ldots,x_n,v) = g(S_n^1(v,v),x_1,\ldots,x_n)$$

y sea z el número de un programa que computa h, entonces or el teorema del parámetro tenemos que $\Phi_z^{(n+1)}(x_1,\ldots,x_n,v) = \Phi^{(n)}_{S^1_n(v,z)}(x_1,\ldots,x_n)$.
Si fijamos $v=z$, entonces

$$\Phi^{(n)}_{S^1_n(v,z)}(x_1,\ldots,x_n) = \Phi^{(n)}_{S^1_n(z,z)}(x_1,\ldots,x_n) = g(S^1_n(z,z),x_1,\ldots,x_n)$$

Finalmente, podemos remplazar $e = S^1_n(z,z)$ en la formula anterior:

$$\Phi_e^{(n)}(x_1,\ldots,x_n) = g(e,x_1,\ldots,x_n)$$

□

Corolario 3.2. *Si $g : \mathbb{N}^{n+1} \to \mathbb{N}$ es parcial computable, existen **infinitos** e tal que*

$$\Phi_e^{(n)}(x_1, \ldots, x_n) = g(e, x_1, \ldots, x_n)$$

DEMOSTRACIÓN

Si, dada una función (parcialmente) computable, hay infinitos programas que la pueden computar. En particular, vale para la función h de la demostración anterior. Por ejemplo, como z es el número de un programa que computa h y le agregamos la instrucciónes $Y \leftarrow Y + 1$ y $Y \leftarrow Y - 1$ al principio, entonces obtenemos un programa equivalente con distinto número que también la computa. Podemos agregar este par de instrucciones la cantidad de veces que queramos y, cada vez, obtendremos un número diferente. □

3.6.6. Teorema del punto fijo

Un **quine** es un programa que cuando se ejecuta, devuelve como salida el mismo programa. Osea si P es un quine tal que $\#(P) = e$, entonces $\Phi_e(x) = e$.

Proposición 3.3. *Hay infinitos e tal que $\Phi_e(x) = e$*

DEMOSTRACIÓN

La demostración sale directamente de aplicar el teorema de la recursión sobre $g : \mathbb{N}^2 \to \mathbb{N}$, $g(z, x) = z$. Por el teorema de la recursión sabemos que existen infinitos programas con número e tal que $\Phi_e(x) = g(e, x) = e$. □

Proposición 3.4. *Sea h una función parcial computable, hay infinitos e tal que $\Phi_e(x) = h(e)$*

DEMOSTRACIÓN

Si consideramos $g : \mathbb{N}^2 \to \mathbb{N}$, $g(z, x) = h(z)$. Por el teorema de la recursión sabemos que existen infinitos programas con número e tal que $\Phi_e(x) = g(e, x) = h(e)$.

Teorema 3.15. *Si $f : \mathbb{N} \to \mathbb{N}$ es computable, existe un e tal que $\Phi_{f(e)} = \Phi(e)$*

DEMOSTRACIÓN

Considerar la función $g : \mathbb{N}^2 \to \mathbb{N}$, $g(z, x) = \Phi_{f(z)}(x)$
Aplicando el Teorema de la Recursión, existe un e tal que para todo x:

$$\Phi_e(x) = g(e, x) = \Phi_{f(e)}(x)$$

□

4. Conjuntos computables enumerables

Parte II

Lógica

5. Lógica Proposicional

6. Sistemas deductivos para lógica propisicional

7. Lógica de primer orden

8. Completitud y compacidad para lógica de primer orden

A. Otras funciones primitivas recursivas:

A.1. Division

$y \text{ div } y = \min_{t \leq x} x * (t+1) > y$

A.2. Divisor

$x|y = (\exists t)_{\leq y}\ x * t = y$

A.3. i-ésimo primo

$p_0 = 2$

$p_{t+1} = \min_{k \leq p_t! + 1}(k > p_t \wedge \text{esPrimo}(p_k)$

B. Otras Macros

B.1. $V \leftarrow 0$

$[A]$ $V \leftarrow V - 1$
IF $V \neq 0$ GOTO A

La macro resta uno a V hasta que sea cero.

B.2. $V \leftarrow k$

$V \leftarrow 0$
$V \leftarrow V + 1$
$V \leftarrow V + 1$
$\vdots$
$V \leftarrow V + 1$

La macro asigna cero a V y luego la incrementa en uno k veces.

B.3. $x = 0$

IF $X \neq 0$ GOTO E
$Y \leftarrow Y + 1$

Este programa recibe X como parámetro. Devuelve 1 si X es igual a cero y devuelve 0 si no.

B.4. $V \leftarrow f(V_1, ..., V_n)$

Dada f una función parcialmente computable queremos asignar a V el resultado de pasarle como parámetro las variables $V_1, \ldots, V_n$.

Como f es parcial computable, sabemos que existe un programa P tal que $f(x_1, \ldots, x_n) = \Psi_P^{(m)}(x_1, \ldots, x_n)$. Asumamos que P usa las variables $X_1, \ldots, X_m$, $Z_1, \ldots, Z_k$ e Y, además de las etiquetas $E, A_1, \ldots, A_l$.

Además, supongamos que para cada instrucción de la forma IF $V \neq 0$ GOTO A_i, la etiqueta referenciada se encuentra dentro del programa P, por lo que la única etiqueta de *salida* es E.

Entonces, si tenemos un programa P_0 en el que queremos almacenar el resultado de una función, solo hace falta renombrar las variables y etiquetas usadas por P a variables y etiquetas que no hayan sido usadas por P_0.

Sea m un número lo suficientemente grande como para que las variables y etiquetas que remplazamos no estén usadas por P_0 entonces, podemos hacer los siguientes remplazos: $Y \to Z_m$, $X_i \to Z_{m+i+1}$, $Z_i = Z_{m+n+1+i}$ y $A_i \to A_{m+i}$.

$$
\begin{array}{ll}
 & Z_m \leftarrow 0 \\
 & Z_{m+1} \leftarrow V_1 \\
 & Z_{m+2} \leftarrow V_2 \\
 & \vdots \\
 & Z_{m+n} \leftarrow V_n \\
 & Z_{m+n+1} \leftarrow 0 \\
 & \vdots \\
 & Z_{m+n+k} \leftarrow 0 \\
 & P \\
[E_m] & W \leftarrow Z_m
\end{array}
$$

Primero *inicializamos* las variables de P. Z_m es donde el programa P (con los remplazos) va a guardar su resultado y Z_{m+1} a Z_{m+n} son las variables que remplazaron a los parámetros y Z_{m+n+1} a Z_{m+n+k} las variables locales. Además, de los remplazos de las etiquetas A_i dentro de P, remplazamos E por E_m.

De esta forma, cuando P termina tanto porque se acabó la lista de instrucciones o porque hizo un salto a E_m asignamos el valor de Z_m a W.

Si quisieramos ejecutar P dentro de un loop, cada vez que termine va a dejar los valores de la última ejecución en las variables que use. Por esta razón es necesario realizar la *inicialización*.

B.5. IF $p(V_1, \ldots, V_n)$ GOTO L

$$
\begin{array}{l}
Z \leftarrow p(V_1, \ldots, V_n) \\
\text{IF } Z \neq 0 \text{ GOTO } L
\end{array}
$$

Usando la macro definida en B.4, asignamos a una variable Z el resultado de ejecutar el predicado p con $V_1, \ldots, V_n$ como variables de entrada. Luego ejecutamos la instrucción IF.

C. Otras definiciones

C.1. La función de Ackerman

Es una función que no es primitiva recursiva.

$$
A(x, y, z) = \begin{cases}
y + z & \text{si } x = 0 \\
0 & \text{si } x = 1 \text{ y } z = 0 \\
1 & \text{si } x = 2 \text{ y } z = 0 \\
y & \text{si } x > 2 \text{ y } z = 0 \\
A(x - 1, y, A(x, y, z - 1)) & \text{si } x, z > 0
\end{cases}
$$

C.1.1. Versión de Robinson & Peter

$$B(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ B(m-1,1) & \text{si } m > 0 \text{ y } n = 0 \\ B(m-1, B(m,n-1)) & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$