Otros formalismos

Autómatas y lenguajes formales

1 de diciembre de 2022

1. Los otros modelos

Ahora que estamos a la distancia y nos parece tan lejana la vida en la facultad (en algunas cosas para bien, en otras no) iniciaré esta nota con una anécdota. Tuve en la facultad un profesor de geometría y álgebra lineal que aún está ahí en el departamento de matemáticas. En su época de estudiante de ciencias la costumbre era estudiar las dos carreras de física y matemáticas (ya ven que en la facultad casi no se da eso), él por ser parte del "hype" entró a ambas carreras, originalmente estudiaba física.

Como era de esperar la demanda de tiempo y trabajo era demasiada, a medio semestre ya estaba flaco, ojeroso y sin ilusiones, no soportó tanto y decidió abandonar una carrera. Lo raro fue que no decidió abandonar la carrera extra, que era matemáticas, prefirió renunciar a física. Sus razones eran las siguientes: la ventaja de la matemática sobre la física es que puedes tomar varios caminos, el que te agrade, el que se te ocurra, y llegar al resultado, más de uno de esos caminos es correcto. En cambio en la física es todo más rígido, hay un sólo camino (por lo regular) y no hay mucha posibilidad para cambiar la ruta.

Esto se debe mucho a la experiencia que tuvo este profesor en ese entonces estudiante, seguro hay áreas de la física que dan libertad. Pero en lo que concuerdo completamente es que en las matemáticas hay un poco más de libertad para enfrentar un problema, más de una persona puede llegar al resultado correcto por caminos distintos y ser válido. Lo mismo sucede en teoría de la computación (que como han podido ver en este y otros cursos tiene un origen en común con las matemáticas), no sólo hay un único modelo computacional.

Ahora, sí yo quisiera hacer otro modelo ¿cómo podría saber que es correcto? '¿Cómo saber que en efecto estos modelos distintos hacen lo mismo? La idea de base es la *computabilidad efectiva*, el rasgo en común que tienen estos modelos. La tesis de Church-Turing (que es más una afirmación) dice:

Tesis de Church-Turing 1 *Toda función es calculable efectivamente sí y sólo sí es calculable por una máquina de Turing.*

Aquí van a decir que ya cayó más pronto un hablador que un cojo, hay una posición preferente a la máquina de Turing ¿no que muy incluyente? Es cierto, la máquina de Turing tiene una posición preferente, de acuerdo al texto por su simplicidad y claridad, pero todos los modelos equivalentes pueden representarse por una máquina de Turing, entonces, como lo hemos hecho en el curso, la máquina de Turing es la que tomamos como paradigma, pero los formalismos que veremos son equivalentes. Si para ustedes no fue tan claro y les queda más

claro por estos modelos, entonces podrán decir que se toma la máquina de Turing por pura formalidad. Lo que veremos en estas opciones es que en algunos casos son más parecidos a un lenguaje de programación de los que ahora usamos.

2. Funciones recursivas μ

Una buena pregunta sería, ¿cuál es el mínimo de funciones necesarias para definir a todas las funciones computables? ¿Cuáles son las funciones de ese conjunto mínimo? De acuerdo a Gödel esas funciones numérico-teóricas $\mathbb{N}^k \to \mathbb{N}$ son:

1. *Sucesor.* La función $\mathbf{s} : \mathbb{N} \to \mathbb{N}$ dadas por $\mathbf{s}(x) = x + 1$ es computable.
2. *Cero.* La función $\mathbf{z} : \mathbb{N}^0 \to \mathbb{N}$ dada por $f() = 0$ es computable.
3. *Proyecciones.* Las funciones $\pi^n_k : \mathbb{N}^n \to \mathbb{N}$ dadas por $\pi^n_k(x_1, ..., x_n) = x_k,\ 1 \leq k \leq n$, son computables.
4. *Composición.* Si $f : \mathbb{N}^k \to \mathbb{N}$ y $g_1, ...g_k : \mathbb{N}^n \to \mathbb{N}$ son computables, entonces también lo es la función $f \circ (g_1, ..., g_k) : \mathbb{N}^n \to \mathbb{N}$ que en la entrada $\overline{x} = x_1, ..., x_n$, da
$$f(g_1(\overline{x}), ..., g_k(\overline{x}))$$
5. *Recursión primitiva.* Si $h_i : \mathbb{N}^n \to \mathbb{N}$ y $g_i : \mathbb{N}^{n+k+1} \to \mathbb{N}$ son computables, $1 \leq i \leq k$, entonces también lo son las funciones $f_i : \mathbb{N}^{n+1} \to \mathbb{N},\ 1 \leq i \leq k$, definidas por inducción mutua de la siguiente manera:
$$f_i(0, \overline{x}) \stackrel{def}{=} h_i(x)$$
$$f_i(x + 1, \overline{x}) \stackrel{def}{=} g_i(x, \overline{x}, f_1(x, \overline{x}), ..., f_k(x, \overline{x})),$$
donde $\overline{x} = x_1, ..., x_n$.
6. *Minimización no acotada.* Si $g : \mathbb{N}^{n+1} \to \mathbb{N}$ es computable, entonces también lo es la función $f : \mathbb{N}^n \to$ que con la entrada $\overline{x} = x_1, ..., x_n$ de al menos y tal que $g(z, \overline{x})$ esté definida para todas las $z \leq y$ y $g(y, \overline{x}) = 0$ si tal y, y está indefinida de otra manera. Esto se denota como:
$$f(\overline{x}) = \mu y.(g(y, \overline{x}) = 0)$$

Algunas de estas funciones suenan demasiado enredadas, pero vamos viendo por pasos. Las primeras cuatro parecen bastante directas, el *sucesor* nos da la posibilidad de construir cualquier número a partir de una base, e incluso nos es útil para definir operaciones más complicadas, como la suma de dos o más números, la multiplicación, el factorial (no se preocupen, lo vamos viendo). La operación lo que hace es sumar un 1 al número que se le dé como entradad a la función.

La función *cero*, $\mathbf{z}$, construye nuestro primer número, el cero, sin necesidad de una entrada, a partir de él podemos construir los demás naturales. La función *proyección*,π^n_k elige el elemento k (esta enumeración va de acuerdo a su posición en la lista) de una lista de n elementos. Y la función *composición* hace

la composición que ya conocemos de dos funciones, así podemos componer *proyección* con *sucesor*, de forma que al elemento k le sumemos un 1 y eso de como resultado.

$$s(\pi_k^n(\overline{x})) = x_k + 1.$$

Para poder definir funciones más complicadas necesitamos de la recursión primitiva, aquí ya empieza a sonar a lenguaje de programación. Veamos como podemos armar la suma con estas funciones.

Vamos a llamarle la función $suma(x)$ y vamos a ponerla en forma de recursión primitiva. Siguiendo los pasos del quinto punto debemos tener una h y una g. Sea $h = \pi_1^1$, la proyección de una lista de un miembro en la posición 1 (no hace nada esta función más que darnos el mismo número que le hemos dado) y sea $g = \mathbf{s} \circ \pi_3^3$ la composición de la función sucesión con la proyección del tercer miembro de una lista de 3. Noten que la función $h : \mathbb{N} \to \mathbb{N}$, va de la dimensión uno a la dimensión uno, $n = 1$, y $g : \mathbb{N}^3 \to \mathbb{N}$, notemos que como $n = 1$, entonces $k = 1$ y así $n + k + 1 = 3$.

¿Porqué se eligieron así las funciones? Para verlo lo ponemos en la forma de la recursión primitiva, *suma* será nuestra función f_i, como $1 \leq i \leq k$ y en nuestro caso $k = 1$ entonces $i = 1$, sólo hay una $f = suma$:

$$\begin{aligned} suma(0, y) &\overset{def}{=} h(y) = \pi_1^1(y) = y \\ suma(x + 1, y) &\overset{def}{=} g(x, y, suma(x, y)) = s(\pi_3^3(x, y, suma(x, y))) = s(suma(x, y)) \end{aligned}$$

Y ahí con recursión se puede llegar a suma de cualquier par de números.

Ahora, a partir de la suma definimos la multiplicación:

$$\begin{aligned} mult(0, y) &\overset{def}{=} z() = 0 \\ mult(x + 1, y) &\overset{def}{=} suma(y, mult(x, y)) \end{aligned}$$

De nueva cuenta hay una recursión inmiscuida para hacer cualquier operación.

Se dan una idea de cómo hacer más cosas ¿cierto? Ahora de tarea, para todxs lxs gamers, hagan el *The Last of us* con puras funciones recursivas, debe quedarles algo como en la figura 1.

Figura 1: *The Last of Us* en 8 bits. Imagen de Rgznsk, tomada de: `https://www.it8bit.com/post/158114971533/the-last-of-us-pixel-art-created-by-rgznsk`

Bueno, después de mi chiste bobo y antes de pasar a cálculo λ, seguro se preguntarán ¿y la sexta función, la minimización no acotada, dónde se usa? Bueno, esta es una operación que como verán en la descripción está relacionada con hasta que punto está definida la función, esto se puede relacionar con el orden y la búsqueda en listas, no coman ansias, ya lo verán en análisis de algoritmos. Su utilidad es un poco menos intuitiva que las demás funciones, así que en este punto es importante que sepan que existe.

Calculo λ

Lisp y sus variaciones (*scheme, common lisp*) son lenguajes de programación muy parecidos al cálculo λ. Este es el histórico, *lisp* es uno de los lenguajes más antiguos y en verdad ahora se considera una familia. *Haskell* es un lenguaje un poco más moderno (un poco menos moderno que ustedes, data de 1990 su primera versión), esto del calculo λ seguro les va a parecer familiar si conocen *haskell* o algún dialecto de *lisp*. Y si son gamers han de saber que el estudio que hace *The Last of Us* inició trabajando con su propio dialecto de *lisp*, que ahora usan menos por cuestiones de portabilidad estando en una empresa más grande, *sony*.

El cálculo λ hace uso de un conjunto de objetos llamados λ-términos y algunas reglas para manejarlos. Por ejemplo, si queremos escribir la función sucesor de las funciones recursivas μ, lo escribiríamos:

$$\lambda x.(x+1),$$

que quiere decir çon la entrada x calcula $(x+1)$", es decir, la función sucesor. Al aplicarlo a un número, digamos 4, se escribiría $(\lambda x.(x+1))4 \to 4+1=5$. La composición sería:

$$\lambda x.f(gx),$$

con la entrada x le aplica g y luego f, lo único que se le da como entrada a esta función es x; g y f ya están definidas dentro. Si quisiéramos darlas también como entradas de la función entonces se escribiría $\lambda f.\lambda g.\lambda x.f(gx)$. ¿Tienen una idea? Si no, no dejen de preguntar. Ahora veremos una versión refinada que llaman el cálculo λ puro.

Calculo λ puro

En esta variante sólo hay variables y operadores para λ-abstracciones y aplicaciones. Para construir un λ-término de manera inductiva a partir de:

- Cualquier variable x es un λ-término
- Si M y N son λ-términos, entonces MN es un λ-término (aplicación funcional, M es la función que está a punto de ser aplicada a la entrada N)
- Si M es un λ-término y x es una variable, entonces $\lambda x.M$ es un λ-término (abstracción funcional, $\lambda x.M$ es la función que con la entrada x computa M).

Esto suena mucho a *haskell* y como el mismo formalismo lo menciona, algo abstracto, vamos revisando.

- La aplicación funcional no es asociativa, y si se escribe sin paréntesis por convención las expresiones se asocian a la izquierda, es decir $MNP = (MN)P$
- Los λ-términos sirven como funciones y datos (recuerden lo de universalidad y autoreferencia), de tal manera para definir la función sucesión debemos decodificarla de alguna manera.
- Si tenemos funciones con más de una variable la podemos definir como:

$$\lambda x_1 x_2 ... x_n.M \stackrel{def}{=} (\lambda x_1.(\lambda x_2.(...(\lambda x_n.M)...))),$$

  los paréntesis mantienen el orden y podemos ver el orden de la abstracción.

¿Cómo realizamos la codificación del punto 2? A la sustitución se le llama una β-reducción y funciona más o menos así: en un término mucho más grande y general se puede tener el subtérmino $(\lambda x.M)N$ puede ser substituido por $M_{[x/N]}$, este término abreviado denota a lo obtenido cuando:

1. renombrando las variables acotadas de M (las y que se dan como entrada a M como un término λy), así ninguna x o N aparezcan como acotadas a M. Así evitamos confundir variables.
2. Sustituyendo N para todas las ocurrencias de x en el término resultante (esta realmente es la sustitución, el anterior sólo es un paso previo).

Como pueden ver en el término abreviado cualquier aplicación se hará sobre M, x y N van en la bolsa. Lo de renombrar variables en unos casos puede verse como una labor directa, por ejemplo en el término $\lambda y.xy$ podemos renombrar y como z y queda $\lambda z.xz$ que realmente no cambia en nada el término, pero si hubiera una doble aplicación nos ayudaría a diferenciar. Pero veamos un ejemplo, la función que duplica la entrada:

$$(\lambda x.xx)z \underset{\beta}{\rightarrow} (xx)_{[x:=z]} = zz$$

Comparando con la β reducción en esquema: M es xx, N es z y λx es ella misma. Una vez que se ha aplicado la β-reducción el término está en su forma normal. La forma normal se traduce directamente a una función de transición en una máquina de Turing. No todos los términos tienen β-reducción.

Ahora convirtiendo las funciones recursivas μ a cálculo λ:

$$
\begin{aligned}
V_{bool} &\stackrel{def}{=} \lambda xy.x \\
F_{bool} &\stackrel{def}{=} \lambda xy.y \\
[M,N] &\stackrel{def}{=} \lambda z.zMN \\
\overline{0} &\stackrel{def}{=} \lambda x.x \\
\overline{n+1} &\stackrel{def}{=} [F,\overline{n}] \\
S &\stackrel{def}{=} \lambda x.[F,x] \\
P &\stackrel{def}{=} \lambda x.xF \\
C &\stackrel{def}{=} \lambda x.\overline{0} \\
Cero &\stackrel{def}{=} \lambda x.xV \\
\pi_i^n &\stackrel{def}{=} \lambda x_1,...,x_n.x_i \\
Y &\stackrel{def}{=} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))
\end{aligned}
$$

Con $\overline{0}$ y $\overline{n+1}$ construimos los números naturales (en las funciones recursivas μ no mostramos el constructor de los número naturales, pero pueden checarlo o intentar hacerlo). S es la función sucesor y por facilidad agregamos la P (predecesor, que también se puede contruir de funciones recursivas μ), V y F los valores booleanos, por ahí también pueden ver la proyección, una función que comprueba si la entrada es igual a cero y da una salida booleana.

¿Para que sirve esa función Y? Es para la recursión. Veamos como definiríamos la recursión primitiva en cálculo λ. Pongamos de ejemplo que tenemos dos funciones $f : \mathbb{N}^k \to \mathbb{N}$ y un grupo de funciones $g_1, ..., g_k : \mathbb{N}^n \to \mathbb{N}$, hacemos la β-reducción para estas funciones, entonces f la pasamos a F y las $g_1, ..., g_k$ a $G_1, ..., G_k$, la composición se escribiría como:

$$\lambda x_1...x_n.F(G_1x_1...x_n)...(G_kx_1...x_n). \tag{1}$$

Con esta base pasamos a la definición de la recursión primitiva (chequen líneas arriba), con las funciones dadas $h : \mathbb{N}^n \to \mathbb{N}$ y $g : \mathbb{N}^{n+2} \to \mathbb{N}$ computables que en β-reducción pasan a H y G. Ahora la función recursiva $f : \mathbb{N}^{n+1} \to \mathbb{N}$ que se define a partir de h y g, con sus respectivas *beta*-reducciones:

$$Y\lambda f.\lambda yx_1...x_n.(Cero\ y)(Hx_1...x_n)(G(Py)x_1...x_n(f(Py)x_1...x_n)).$$

Con esta estructura podemos construir la suma de forma recursiva en cálculo λ (nuestra función f será $SUMA$):

$$
\begin{aligned}
H &\stackrel{def}{=} \lambda x_1.x_1 \\
G &\stackrel{def}{=} \lambda z,x_1,x_2.S((\lambda x_1,x_2,x_3.x_3)zx_1x_2) \\
SUMA &\stackrel{def}{=} Y\lambda f.\lambda yx.(Cero\ y)(Hx)(G(Py)x(f(Py)x)).
\end{aligned}
$$

Esto lo pueden ver sólo de sustituir en la estructura de arriba, tomando en cuenta que $k = 1$ y $n = 1$, es decir, sólo hay una x.

De manera similar pueden convertir la minimización no acotada.

Lenguaje IMP

El lenguaje IMP es un modelo sencillo, aún con partes de abstracción, de un lenguaje de programación de ato nivel. Se pueden reconocer estructuras como el if y el $while$ presentes en la mayoría de lenguajes de programación usados.

Se define como una gramatica independiente de contexto, pero es un lenguaje Turing completo

$$
\begin{aligned}
&P \to skip \mid X := A \mid (P; P) \mid (if\ B\ then\ P\ else\ P) \mid (while\ B\ do\ P)\\
&A \to Z \mid X \mid (A + A) \mid (A - A) \mid (A \times A)\\
&B \to v \mid f \mid (A = A) \mid (A < A) \mid \neg B \mid (B \wedge B) \mid (B \vee B)\\
&X \to x_N\\
&N \to 0 \mid C\\
&C \to 1S \mid 2S \mid ... \mid 9S\\
&S \to 0S \mid 1S \mid ... \mid 9S \mid \epsilon\\
&Z \to N \mid -C
\end{aligned}
$$

donde podemos identificar las expresiones aritméticas(A), las expresiones booleanas (B), la producción d elos números enteros Z, las localidades de memoria (X) y los comandos del lenguaje P.

En este caso no es necesario definir la suma desde una función más sencilla, pero lo que sí hay que definir es la recursión, eso se puede hacer a partir de **if** y **while**.

Por ejemplo, si para el caso cero queremos evaluar una función h lo hacemos como se puede ver en el algoritmo 1

Algoritmo 1 Evaluación diferenciada

```
1: inicializaciones
2: if Condición inicial then
3:     h es evaluada
4: else g es evaluada
5: end if
```

Lo pongo muy platicado, pero ya les toca a ustedes llenar los huecos.

Pero por poner un ejemplo ya sin recursión primitiva podemos definir la operación factorial de la forma como se ve en el algoritmo 2.

Programas While

De nueva cuenta nos movemos a terrenos más conocidos de la programación, para ello definimos un lenguaje de programación sencillo.

Algoritmo 2 Función factorial de n

1: $x_0 := 1$;
2: **while do**$(x_1 < n)do$
3: $\quad x_1 := (x_1 + 1)$;
4: $\quad x_0 := (x_0 \times x_1)$
5: **end while**

1. asignación simple: $x := 0,\ x := y + 1,\ x := y$
2. composición secuencial: $p : q$
3. condicional: $if\ x < y\ then\ p\ else\ p$
4. for en ciclos: $for\ y\ do\ p$
5. $while$ en ciclos: $while\ x < y\ do\ p$

con las relaciones $<, >, \leq, \geq, =, \neq$ intercambiables en los puntos 3 y 5.

Ya saben como funciona el for y $while$, que incluso hasta cierto grado son intercambiables salvo que los programas con for se detienen y el $while$ puede no detenerse y entrar en un ciclo infinito. Eso suena a una máquina de Turing bastante general y por eso nos centraremos en él.

Un estado o ambiente σ es una asignación de un número natural a cada variable en el conjunto **Var**. Al conjunto de los ambientes le llamaremos **Env**. Al ejecutar el programa las variables irán cambiando, por lo que los ambientes también, pero todo dentro de los conjuntos definidos. En un programa las variables van cambiando, las asignaciones son dinámicas, entonces un programa es una función parcial que va de un ambiente a otro $[[p]] : \mathbf{Env} \to \mathbf{Env}$. Si el programa se detiene el ambiente final será $[[p]]\sigma$, de no detenerse esto no estará definido.

En este formalismo un programa $while$ se definiría a partir de $\sigma \in \mathbf{Env}$, $x \in \mathbf{Var}$ y $a \in \mathbb{N}$, siendo el ambiente $\sigma[x \leftarrow a]$ (el ambiente idéntico σ excepto por el valor x que es a). Formalmente:

$$\sigma[x \leftarrow a](y) \stackrel{def}{=} \sigma(y) \textbf{ sí y no es x}$$
$$\sigma[x \leftarrow a](y) \stackrel{def}{=} a$$

Extra

Ahora les van recomendaciones de libros que pueden descargar desde internet:

Editorial Tumbona. Es una editorial independiente mexicana dedicada a el ensayo en su mayor parte. Es muy recomendable su colección **Versus** de ensayos cortos, entre ellos está un compendio llamado *Contra el Copyright* con textos de Richard Stallman y la colectividad Wu Ming. De ahí debió nacer la idea de compartir varios de sus títulos, no sólo de esa colección. Muy recomendable *Contra el copyright* y también *Contra la televisión* de Heriberto Yépez, autor tijuanense. Quizá ya hoy en día la televisión esté muy superada, pero es un libro

interesante donde el autor desarrolla sus ideas sobre el daño que la televisión le ha hecho a la sociedad mexicana, le llama la *telefísica*. Echen un ojo al catálogo y llévense el que les agrade, como decía, los ensayos de **Versus** son cortos por lo que no será tan incómodo leerlos con pausas en una pantalla. La página es: `http://tumbonaediciones.com/descarga-de-libros/`.

Crunch editores. Era una editorial mexicana dedicada a editar puros libros electrónicos, en un principio en pdf, después ya en epub. Su página oficial ya no existe y los libros sólo se encuentran en el *internet archive*, por suerte siempre fueron gratis y de libre distribución. Les recomiendo *La gente se droga* de otro escritor de Tijuana, Rafa Saavedra. Es un libro muy corto pero bastante experimental, sin perder contenido. Echen un ojo al catálogo, son libros cortos. En el internet archive la búsqueda: `https://archive.org/search.php?query=crunch%20editores`.

Librería del Partido Interdimensional Pirata. Seguro ya conocen al partido Pirata, con sedes en varios países, desde el principio han estado a favor de compartir libremente la cultura. En su librería hay textos políticos en su mayoría, algunos zines, todo en descarga libre y en varios formatos. Aquí sí no puedo recomendarles nada, he ojeado varios de estos libros, los he empezado a leer, conozco otros de algunos de los autores, pero no puedo decirles más. Si les es útil, quieren conocer estas ideas y seguro si las comparten les dará gusto saber que no son los únicos que piensan así. La página: `https://utopia.partidopirata.com.ar/`.

Hay muchos libros que se encuentran en dominio público y valen la pena, es fácil encontrarlos en distintas ediciones electrónicas. *El Juguete Rabioso* del escritor argentino Roberto Arlt es una novela centrada en la vida de un joven a principios del siglo XX, sin mucho futuro, algo molesto con la vida. Anterior a John Fante, me parece una novela muy cercana a *Espera la Primavera Bandinni*, pero en versión latinoaméricana. La historia no es parecida, pero el estilo que posteriormente llamaron realismo sucio me parece ya se notaba desde Arlt. Es un libro más extenso, quizá si llama su atención pueda convenir conseguirlo impreso (debe haber edición ya en editoriales muy baratas), pero por si gustan y tienen la posibilidad pueden bajar la versión escaneada de aquí: `https://archive.org/details/RobertoArltElJugueteRabioso`.

Referencias

[1] Kozen, Dexter C. “Automata and Computability” Springer (1997)