Cálculo λ y programación funcional

Programación funcional para la física computacional

26 de noviembre de 2023

Máquinas de Turing

La máquina de Turing es uno de los modelos de computación más usados, es lo más cercano a una computadora (como la que usaron para bajar este pdf, y quizá hasta para leerlo), pero no es el único modelo. Ahora lo mencionó sólo para dar entrada a la idea del cálculo λ, para entender las cosas. No sé si será el método más pedagógico, o el más sencillo, pero es el estándar en la mayoría de libros del tema (además de que Turing es el más famoso de los matemáticos que trabajaron inicialmente en el área). Los modelos son equivalentes, así que no perderemos generalidad.

Una máquina de Turing se compone de un conjunto finito de estados Q, una cinta semi-infinita limitada por la izquierda con el símbolo $\vdash$ e ilimitada por la derecha (este límite izquierdo es para saber donde empieza la cinta) y una cabeza que puede moverse a izquierda y derecha, capaz de leer y escribir caracteres en la cinta[1]. Las palabras de entrada, de longitud finita, se escriben sobre la cinta de izquierda a derecha (como escribimos nosotros). Al terminar la palabra de entrada en el resto de casillas de la cinta para distinguir que no contienen caracter alguno está pre-escrito el símbolo $\sqcup$. Un esquema ejemplificando esta disposición se muestra en la figura **??**.

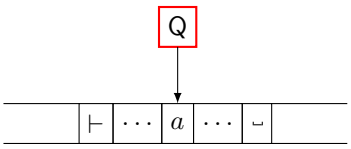

Figura 1: Esquema mecánico de una máquina de Turing.

Podemos dar una definición formal:

Definición 1 *Una máquina de Turing determinista, de cinta única es una 9-tupla (yo lo traduciría como un noneto o enéada, pero quizá no es la terminología) descrita como:*

$$M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r)$$

con:

[1] Son muy jóvenes para recordar los casetes quizá, aunque por ahí hay unos intentos retros de revivirlos, pero si alguna vez han visto uno con su respectivo reproductor podrán notar que cuenta con una cinta magnética de color café o negro, que es leída por una cabeza también magnética. Era un formato de no muy buena calidad, de riesgo pues un imán de potencia suficiente podía dañar la cinta y los carretes podían provocar accidentes como enrollarse o atorarse. Hubo computadoras que usaban casetes para leer programas, para una crónica al respecto echen un ojo a: `https://www.xataka.com/historia-tecnologica/cuando-los-videojuegos-venian-en-cassette-y-habia-que-rebobinarlos-para-poder-jugar`

- Q *es el conjunto finito de estados*
- Σ *es el alfabeto de entrada (finito)*
- Γ *es el alfabeto de cinta (finito), con* $\Sigma \subseteq \Gamma$
- $\vdash \in \Gamma - \Sigma$ *el símbolo de inicio de la cinta*
- $\sqcup \in \Gamma - \Sigma$ *el símbolo de espacio en blanco*
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\to, \leftarrow\}$ *la función de transición*
- $s \in Q$ *el estado inicial*
- $t \in Q$ *el estado de aceptación*
- $r \in Q$ *el estado de rechazo.*

Un ejemplo

Sea la máquina de Turing que decida el lenguaje $A = \{0^{2^n} | n \geq 0\}$, es decir, el lenguaje con puras cadenas de 0's que su longitud sea una potencia de 2.

	$\vdash$	0	x	$\sqcup$
s	$(s, \vdash, \to)$	$(q_1, \sqcup, \to)$	$(r, -, -)$	$(r, -, -)$
q_1		$(q_2, x, \to)$	$(q_1, x, \to)$	$(t, -, -)$
q_2		$(q_3, 0, \to)$	$(q_2, x, \to)$	$(q_4, \sqcup, \leftarrow)$
q_3		$(q_2, x, \to)$	$(q_3, x, \to)$	$(r, -, -)$
q_4		$(q_4, 0, \leftarrow)$	$(q_4, x, \leftarrow)$	$(q_1, \sqcup, \to)$

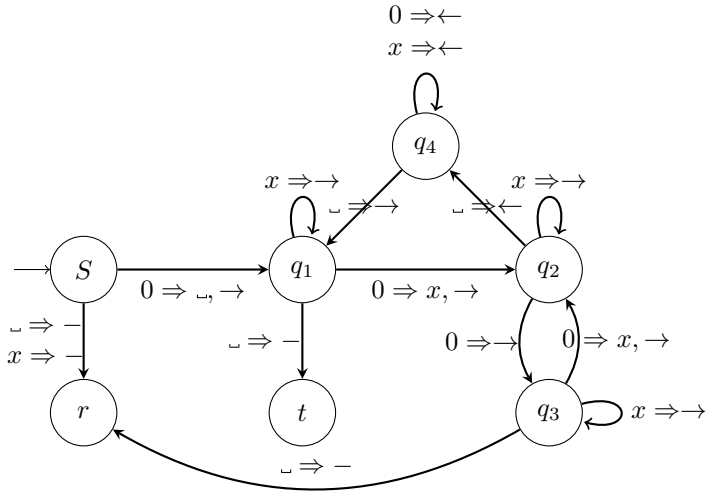

Figura 2: Diagrama de estados de la máquina de Turing descrita.

Otros modelos computacionales

El modelo de máquina de Turing ajusta muy bien para cuando se trabaja de manera teórica con computadoras, pero quizá pueda parecer demasiado mecánico y no se parece a como operamos con computadoras hoy en día. Existen más modelos que sonm equivalentes, sabemos de esa equivalencia por la **tesis Church-Turing**.

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

En esos otros modelos se encuentran las funciones recursivas μ, relacionadas dierctamente al trabajo de Gödel al preguntarse ¿cuál es el mínimo de funciones necesarias para definir a todas las funciones computables?

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:
$$f(\overline{x}) = \mu y.(g(y, \overline{x}) = 0)$$

Cálculo λ

Ya en la introducción habíamos mencionado una forma de ver a las funciones matemáticas como cajas negras, figura . Ahora vamos un paso adelante a definir de manera abstracta lo que es una función, una *abstracción funcional* y como aplicamos las funciones desde una perspectiva abstracta y meramente matemática.

Las características del cálculo λ pueden listarse:

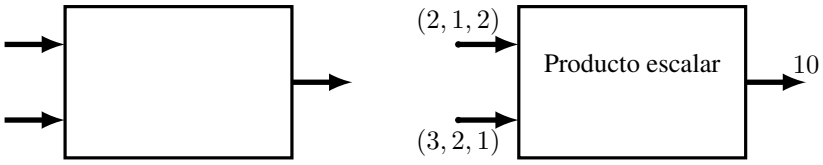

Figura 3: Una función representada como caja negra.

- Solo depende de funciones. Si no se puede escribir como función, no se puede incluir.
- No tiene estado o efectos laterales
- El orden de evaluación no es relevante
- Todas las funciones son unitarias, sólo toman un argumento

El cálculo λ fue iniciado en la década de los 30s del siglo XX por Alonso Church. No fue la única persona trabajando en el área, Haskell Curry trabajo en la lógica combinatoria, Gödel en las funciones recursivas μ en las que podemos encontrar algunas similitudes, Emil Post en el sistema canónico de Post, o sistema de reescritura y Alan Turing con las máquinas de Turing. Todos ellos de alguna manera trataron de definir qué era un algoritmo y darle una estructura formal[2]. Las funciones recursivas μ y las máquinas de Turing ya las vimos de rápido, en lo que sigue nos centraremos en la parte del trabajo de Church y Curry. Pero para ser justos hay que mencionar como funciona el sistema canónco de Post.

Se parte de una cantidad finitamente grande de cadenas de caracteres que serán manipuladas, de manera repetida son transformadas al aplicar un conjunto finito de reglas, formando un lenguaje regular. Al ser Turing completo, es equivalente a trabajar con máquinas de Turing. Ya sólo se estudia con interés histórico pues hay formalismos que incluyen las ideas de Post más usados hoy en día.

La idea de Post al crear el cálculo λ era demostrar que el problema de la decisión de Hilbert no era resolvible por la aritmética de Peano. Lo que obtuvo fue un sistema para estudiar las matemáticas de una forma más general.

Con las reglas antes mencionadas sólo resta decir que para construir todas las funciones matemáticas en el cálculo λ basta usar tres operaciones:

- Crear funciones que recibirán variables
- Asociar una variable a la expresión
- Aplicar una función a un argumento

¿Porque la diferencia entre asociar una variable y aplicar a un argumento? Tal como cuando definimos una función matemática del estilo $f(x) = x^2 - 3x$ las variable x es más un marco donde eventualmente puede entrar un valor, al evaluar la ecuación en un número es que estamos aplicando la función a un argumento. Las variables son términos λ y son la base de las definiciones deductivas del cálculo λ.

[2]Como vimos en la sección pasada los algoritmos existen incluso desde hace 2000 años, pero no había una teoría formal de ellos.

En el trato normal de las variables en el cálculo λ los tipos no entran en juego, está des-tipado (no sé si así se dice, pero suena cagado).

Imaginemos que se tienen dos términos λ M y N, MN también es un término λ, podría ser por ejemplo que M es una función y N una entrada (podría ser también una función). La manera más formal de escribirlo es $(M)N$ donde se dice que N es una entrada aplicada a la función M. El paréntesis es operador *aplicación* y se asocia a la izquierda (MN)P.

Pero realmente el orden de la aplicación es de izquierda a derecha, por ejemplo, si tenemos $\lambda x.E(x)$ lo que quiere decir es que con la entrada x la función calcula $E(x)$.

Veamos un ejemplo concreto. Se tiene:

```
inc(x) = x+1
```

Lo que quiere decir que para incrementar x se le suma 1

```
(x) -> x+1
```

Pero usando ya directamente la abstracción

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

Si ahora tenemos la función

```
sumcuad(x,y) = x^2+y^2
```

Que se traduce a

```
(x,y)-> x^2+y^2
```

¿Cómo se pondría en la abastracción?

$$\lambda x.(\lambda y.(x^2+y^2))$$

¡Muy bien! ¿Y ahora como se evalúa (aplicar a un argumento)? Para el caso del incremento al número 8:

$$\lambda x.(x+1)8$$

Vamos a un caso más abstracto, sea la composición de las funciones f y g

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

Pero en ese caso solo estamos dando como entrada a x ¿y si quiero dejarlo como el esqueleto de una composición done también puedo darle como entrada las funciones f y g?

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

Con ese esqueleto ahora hagamos la composición de la función sucesor:

$$\begin{aligned}
&(\lambda f.\lambda g.\lambda x.f(gx))(\lambda y.(y+1))(\lambda z.(z+1)) \\
&\rightarrow (\lambda g.\lambda x.((\lambda y.(y+1))(gx)))(\lambda z.(z+1)) \\
&\rightarrow \lambda x.((\lambda y.(y+1))((\lambda z.(z+1))x))) \\
&\rightarrow \lambda x.((\lambda y.(y+1))(x+1)) \\
&\rightarrow \lambda x.((x+1)+1)
\end{aligned}$$

Calculo λ *tipado*

Una variante del cálculo λ admite que se especifique el tipo de las variables a dar como argumento, así las funciones definidas se restringen al dominio dado.

Si en la función incremento sólo queremos operar sobre naturales

$$\lambda x : \nu.(x + 1)$$

Eso es el estilo Church, se infiere que al ser la entrada naturales la salida son naturales, pero el estilo Curry hace que toda la función sea definida dentro de los naturales

$$(\lambda x.(x + 1))\nu \to \nu$$

Si hay más de una variable la versión Post sería

$$\lambda x : \nu.(\lambda y : \nu.(x^2 + y^2))$$

Y la versión Curry

$$\lambda x.(\lambda y.(x^2 + y^2)) : \nu \to \nu \to \nu$$

Operaciones de reducción

Vamos ahora a expresar una función λ en su forma más simple y pura

- **Reducción** α: Renombrar variables. Se dice que dos funciones son $\alpha equivalentes$ cuando lo único que cambia es el nombre d ela variable ($\lambda x.(x+1)$ y $\lambda a.(a+1)$).
- **Reducción** β: Remplaza variables por argumentos. Como ya hicimos líneas arriba.
- **Reducción** η: Un aso antes para asegurar la reducción β

Un ejemplo de α equivalencia son la pareja de funciones:

$$\lambda x.(x^2 + 3x)$$
$$\lambda a.(a^2 + 3a),$$

Si cambiamos el nombre de la variable en ambos casos, tenemos la misma función. Siempre es posible renombrar variables mientras no se use el nombre de una ya utilizada.

La reducción β ya viomos un ejemplo, pero para no dejar damos un ejemplo evaluando la función en un natural:

$$\begin{aligned}\lambda x.(x^2 + 3x)(2)&\\ (2^2 + 3(2))&\\ =10,&\end{aligned}$$

y evaluando en optra función, a final de cuentas variables, constantes y funciones todas son λ términos y valen lo mismo para la β reducción

$$\lambda x.(x^2 + 3x)(\lambda y.(y + 1))$$
$$((y + 1)^2 + 3(y + 1)),$$

La reducción η es un paso más abstracto, de cierta manera se asegura de que la forma de la función λ es correcta y está lista para una β reducción. A este punto no es necesario ahondar mucho en ello.

Programación funcional en *Python*

Una de las características de la programación funcional a diferencia de la programación estructurada o la orientada a objetos es que funciona a partir de declaraciones en lugar de procedimientos paso a paso. Es decir, le decimos a la máquina que hacer pero no le indicamos como hacerlo. Así funciona Haskell por diseño, con la ya comentada ventaja de los datos inmutables (evitando los problemas de flujo de datos en concurrencia, como mencionamos la sección pasada).

¿Cómo es eso de los datos inmutables? Vemos en python:

```
x=1
oldID=id(x)
x=x+1
id(x)==oldID
```

El valor ha mutado, pues lo hemos obligado a hacerlo así, pero en un caso más amplio si no somos cuidadosos podría provocar errores en nuestro programa.

La falta de estado, como sucede en Haskell, conlleva la desventaja de la falta de memoria.

Funciones en Haskell

En Haskell existen dos opciones para definir funciones, de forma *curriada* y *no curriada* (por Haskell Curry, no porque le pongamos ese condimento de la comida oriental).

La definición de una función *no curriada*

```
suma (x,y) = x+y.
```

A partir de esa función suma definamos la función sucesor:

```
suc (x) = suma (x,1).
```

Si esto mismo trataramos de hacer en $python$, para definir una operación que deja sin cambio una variable, se tiene que hacer:

```
def DoChange(x, y):
    x = x.__add__(y)
    return x

x = 1
print(x)
print(DoChange(x, 2))
print(x)
```

Es una definición especial de la suma que lo permite, pero no es la norma, por ejemplo:

```
def DoChange(aList):
    aList.append(4)
    return aList

aList = [1, 2, 3]
print(aList)
print(DoChange(aList))
print(aList)
```

Regresando a Haskell, definamos las operaciones de suma e incremento de forma *curriada*, es decir, definimos una función que sólo necesita un argumento y regresa otra función. Cierrra **GHCi** y vuelve a abrir. Ahora define:

```
  suma x y  = x+y
```

Aquí parece que solo quitamos los paréntesis, pero no. Para notar la diferencia ahora definimos la función incremento como:

```
  inc = suma 1
```

Se puede usar *curry* y *uncurry* para pasar de una a la otra. Aprovecha para jugar con *map*.

```
  suma_u = uncurry suma
```

Como haríamos esto mismo en $python$:

```
  def suma(x,y):
    return x+y
```

Para definir el incremento en uno:

```
  def inc(x):
    return suma (x,1)
```

Pero apliquemos lo visto anteriormente de cálculo λ para ambos lenguajes de programación. En Haskell ya vimos un poco pero ampliemos, para definir una función tenemos tres formas equivalentes

$$\begin{aligned} suma\ x &= \backslash\, y- > x + y \\ suma &= \backslash\, x- > (\backslash y- > x + y) \\ suma &= \backslash\, x- > \backslash y- > x + y. \end{aligned}$$

Noten que es como una definición de funciones λ, en lugar de λ se usa $\backslash$ y se sigue la misma estructura. De esta forma puede definirse la suma sin argumentos pero dentro de la definición están los λ términos necesarios para aceptar variables.

Al verlo pueden pensar que so no tiene el mayor chiste, más parece que nos complicamos la vida, pero no, esto tiuene un sentido más. ¿Qué tal que quisiéramos definir un nuevo operador, llamémosle $+=$[3]?

$$\begin{aligned} (+=) &= \backslash\, x- > \backslash y- > x + y \\ (+=) &= \backslash\, x\ y- > x + y \\ 3 &+= 5. \end{aligned}$$

[3] Ahora no tiene mucha diferencia al operador $+$ salvo que nosotros lo definimos, pero imaginen aplicar esto para una función más complicada que puede ser evaluada con un simple operador.

¿Cómo podemos definir funciones λ en $python$? Aunque para este lenguaje de programación el cálculo λ no está directamente aplicado, está de alguna forma desde sus huesos. Por las características de ser más directamente aplicable. más sencillo, esto sirve más como un caso ilustrativo. Las dos formas de definir una función con cálculo λ en $python$:

$$
\begin{aligned}
suma =& lambda\ x, y : x + y \\
& suma(3, 2) \text{ la forema de llamar a la función} \\
suma =& lambda\ x : lambda\ y : x + y \\
& suma(3)(2) \text{ llamando a la función,}
\end{aligned}
$$

vea que las diferentes definiciones en este caso implican llamr de forma distinta a la función. En la definición λ se escribe $lambda$.

Referencias

[1] Thompson, Simon J.. "Haskell - the craft of functional programming." International computer science series (1996).

[2] Mueller, John P. "Functional programming (for dummies)" John Willey & Sons Inc. (2019).

[3] Kozen, Dexter C. "Automata and Computability" Springer (1997)