Lenguajes de Programación 2023-1
Nota de clase 5: Cálculo Lambda
Funcional (λ)

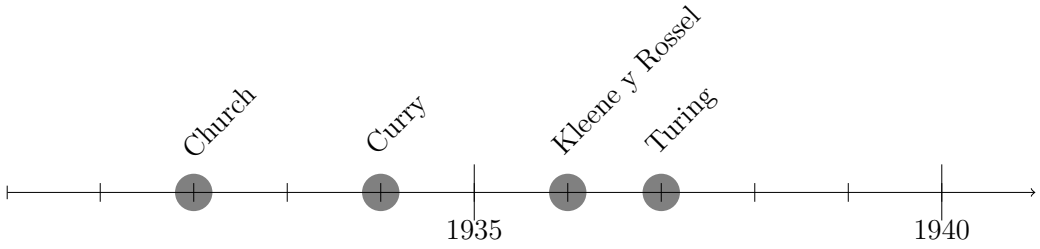
Javier Enríquez Mendoza

15 de septiembre de 2022

Hasta este momento hemos estado trabajando con un lenguaje muy sencillo que únicamente incluye expresiones aritméticas y variables. Este lenguaje si bien es importante para el estudio de los conceptos vistos hasta ahora, no es un lenguaje de programación.

El objetivo final de este curso es estudiar las características de los lenguajes de programación clasificados por el estilo de programación que siguen. En esta nota de clase se presenta un sistema que sirve como núcleo para el diseño y el estudio de los lenguajes de programación del paradigma funcional, este sistema recibe el nombre Cálculo Lambda.

1 Un poco de historia



En 1932 Alonzo Church en colaboración con sus alumnos Stephen Kleene y Barkely Rosser, desarrollaron un sistema basado en funciones y lógica, con el objetivo de que sirviera como fundamento para la matemática. Posteriormente, en 1936 Kleene y Rosser probaron que dicho sistema era inconsistente por lo que no servía su propósito original. Sin embargo, el sub-sistema encargado exclusivamente del manejo de funciones resulto ser un excelente modelo teórico para la definición de funciones computables. Este sub-sistema en la actualidad recibe el nombre de Cálculo Lambda.

En 1936 Kleene y Rosser también probaron que todas las funciones recursivas podían ser representadas en Cálculo Lambda.

Por otro lado en 1937 Alan Turing demostró que el poder de expresividad de una Máquina de Turing era el mismo que tenía el Cálculo Lambda, es decir, que ambos sistemas podían expresar

exactamente los mismos cómputos. Y esta equivalencia es la que recibe el nombre de la Tesis de Church-Turing.

En 1934 Haskell Curry desarrolla la lógica combinatoria, basada en el Cálculo Lambda, con la diferencia de que en este sistema todas las expresiones pueden representarse mediante composiciones de tres combinadores.

Representar los cómputos mediante el uso de funciones o expresiones del cálculo lambda, llamadas λ términos, dio origen a un estilo de programación en el que las funciones son el elemento principal, la *programación funcional*.

2 Sintaxis

Parte importante de la popularidad del Cálculo Lambda se debe a su simplicidad. En este sistema se tienen simplemente tres tipos de expresiones: variables, abstracciones lambda y aplicaciones.

Definición 2.1 (Sintaxis concreta del Cálculo Lambda)**.** La sintaxis concreta del Cálculo Lambda se da con la siguiente definición inductiva, sobre el juicio $l\ \lambda\mathsf{term}$ que indica que l es un término lambda, es decir, l es una expresión válida en el Cálculo Lambda.

$$\frac{x\ \mathsf{var}}{x\ \lambda\mathsf{term}}\ \mathsf{var} \qquad \frac{x\ \mathsf{var} \quad e\ \lambda\mathsf{term}}{\lambda x.e\ \lambda\mathsf{term}}\ \mathsf{abs} \qquad \frac{e_1\ \lambda\mathsf{term} \quad e_2\ \lambda\mathsf{term}}{e_1\, e_2\ \lambda\mathsf{term}}\ \mathsf{app}$$

Variables Se tiene un conjunto infinito de variables, usualmente denotadas por las últimas letras del alfabeto $w, x, y, z, \ldots$, las variables son un término lambda.

Abstracción lambda Una expresión de la forma $\lambda x.e$ es llamada una abstracción lambda o simplemente abstracción, en donde x es la variable de la abstracción y e es el cuerpo. La idea intuitiva es que la abstracción se pueden pensar como una función anónima que reciben como argumento a x y su cuerpo es e. De la misma forma que en la sintaxis de orden superior, el punto denota un ligado de la variable x en el cuerpo e.

Aplicación Una expresión de la forma $e_1\, e_2$ es llamada una aplicación. La idea es que estas expresiones denotan la aplicación de funciones en donde e_1 es la función y e_2 es el argumento. La aplicación asocia a la **izquierda**, por lo que la expresión $e_1\, e_2\, e_3$ significa $(e_1\, e_2)\, e_3$.

Ejemplos (Términos Lambda)**.** A continuación se presentan algunos ejemplos de expresiones del Cálculo Lambda.

Abstracción Lambda :

- $\lambda x.x$
- $\lambda x.y$
- $\lambda x.\lambda y.xy$
- $\lambda x.\lambda y.xyx$
- $\lambda x.xx$

En estos ejemplo la expresión $\lambda x.x$ define la función identidad, es decir, la función que regresa el mismo parámetro que recibe y la expresión $\lambda x.y$ es la función constante que sin importar que recibe regresa y.
La expresión $\lambda x.\lambda y.xy$ denota una función de dos parámetros x y y. Aunque la especifica-

ción de la sintaxis no permite definir funciones de dos o mas argumentos se pueden anidar abstracciones para representar estos caso, en donde cada abstracción define un parámetro de la función, a esta técnica se le conoce como *currificación* en honor a Haskell Curry. Por simplicidad se puede escribir la abstracción como $\lambda xy.xy$, aunque se trata de un abuso de notación.

Aplicación :

- xy
- $(\lambda x.y)x$
- $(\lambda x.x)(\lambda y.y)$
- xyz
- $x(\lambda x.y)$

Dado que en la abstracción $\lambda x.e$ el operador λ liga a la variable x en e se mantiene la noción de variables libres y ligadas introducida anteriormente con las expresiones `let`. También se puede definir la noción de α-equivalencia, de forma similar a la vista con sintaxis de orden superior.

Definición 2.2 (α-equivalencia)**.** En Cálculo Lambda, dos lambda términos e_1 y e_2 son α-equivalentes si y sólo si solo se diferencian en el nombre de las variables de ligado. Por ejemplo, las expresiones:

$$\lambda x.x \qquad \lambda z.z$$

son α-equivalentes y se denota como $\lambda x.x \equiv_\alpha \lambda z.z$

3 Semántica Operacional

Definición 3.1 (Semántica operacional del Cálculo Lambda)**.** La semántica operacional del Cálculo Lambda está dada por la siguiente regla conocida como β reducción.

$$(\lambda x.t)\, s \to_\beta t[x := s]$$

En esta regla, a la expresión $(\lambda x.t)\, s$ se le llama redex (del inglés reducible expression), mientras que a la expresión $t[x := s]$ se le llama reducto . Se observa entonces que todo paso de evaluación es simplemente una sustitución definida de la manera usual mediante el uso de α-equivalencia para evitar la captura de variables libres:

- $x[x := r] = r$.
- $y[x := r] = y$ si $x \neq y$.
- $(ts)[x := r] = t[x := r]s[x := r]$.
- $(\lambda y.t)[x := r] = \lambda y.t[x := r]$ donde $y \notin FV(r)$.

Ejemplo 3.1 (Evaluación de expresiones del Cálculo Lambda)**.** Se evalúa la siguiente expresión haciendo uso de la regla de β reducción.

$$\begin{array}{rl} & (\lambda x.\lambda y.xy)(\lambda x.x)z \\ \rightarrow_\beta & (\lambda y.xy)[x := \lambda x.x]z \\ = & (\lambda y.(\lambda x.x)y)z \\ \rightarrow_\beta & (\lambda y.(x[x := y])z \\ = & (\lambda y.y)z \\ \rightarrow_\beta & y[y := z] \\ = & z \end{array}$$

Se puede observar que el proceso de evaluación termino arrojando como resultado la variable z sobre la cual ya no se puede aplicar la regla de β reducción.

Definición 3.2 (Forma Normal)**.** Se dice que un λ término está en forma normal si no existe otro término e' tal que $e \rightarrow_\beta e'$.
De esta forma, si $e \rightarrow_\beta^* e_n$ y e_n está en forma normal, decimos que e_n es la forma normal de e.

4 Definibilidad Lambda

En el Cálculo Lambda se pueden dar definiciones de expresiones usuales en los lenguajes de programación como lo son las expresiones aritméticas, booleanas o estructuras de datos, así como funciones para trabajar con éstas. En esta sección se muestran algunas de estas definiciones.

4.1 Booleanos

Comencemos dando una definición de los Booleanos en el Cálculo Lambda, para esto se representan las constantes booleanas como funciones con dos argumentos de la siguiente forma:

Verdadero

$$\mathsf{true} =_{def} \lambda x.\lambda y.x$$

Falso

$$\mathsf{false} =_{def} \lambda x.\lambda y.y$$

Con esta definición de conjunto de Booleanos se pueden dar definiciones para operadores sobre ellos, como las siguientes:

El operador `if`

$$\mathsf{if} =_{def} \lambda v.\lambda t\lambda f.vtf$$

Negación

$$\mathsf{not} =_{def} \lambda z.z\,\mathsf{false}\,\mathsf{true}$$

Conjunción

$$\mathsf{and} =_{def} \lambda x.\lambda y.xy\,\mathsf{false}$$

Con estas definiciones es fácil comprobar que las siguientes propiedades son ciertas:

- $\mathsf{if\ true}\ e_1\ e_2 \rightarrow^*_\beta e_1$
- $\mathsf{if\ false}\ e_1\ e_2 \rightarrow^*_\beta e_2$
- $\mathsf{not\ true} \rightarrow^*_\beta \mathsf{false}$
- $\mathsf{not\ false} \rightarrow^*_\beta \mathsf{true}$
- $\mathsf{and\ false}\ b \rightarrow^*_\beta \mathsf{false}$
- $\mathsf{and\ true}\ b \rightarrow^*_\beta b$

Es decir, que los operadores funcionan adecuadamente.

4.2 Expresiones aritméticas

Definamos las expresiones aritméticas en el Cálculo Lambda, para esto se requiere de una representación de los números naturales sobre los cuales se puedan definir estas expresiones.

4.2.1 Numerales de Church

El Cálculo λ permite modelar operaciones aritméticas. Para poder usar estas operaciones, definimos una representación de los números naturales mediante abstracciones lambda. Esta representación recibe el nombre de Numerales de Church.

La idea intuitiva para representar números naturales es construirlos como una función con dos argumentos z que hace referencia al cero y s que modela la función sucesor, por lo que se representa el natural n aplicando n veces s a z. De esta forma los naturales quedan definidos como sigue:

- $\bar{0} =_{def} \lambda s.\lambda z.z$
- $\bar{1} =_{def} \lambda s.\lambda z.sz$
- $\bar{2} =_{def} \lambda s.\lambda z.s(sz)$
- $\bar{n} =_{def} \lambda s.\lambda z.\underbrace{s(\ldots(s}_{n\ veces}\ z)\ldots)$

A partir de esta representación de los números naturales se pueden definir funciones aritméticas que operen con ellos.

4.2.2 Funciones aritméticas

Las operaciones sobre naturales se definen como sigue:

Sucesor

$$\mathsf{suc} =_{def} \lambda n.\lambda s.\lambda z.s(nsz)$$

para calcular el sucesor de cero, tenemos:

$$
\begin{array}{rl}
 & \mathsf{suc}\,\bar{0} \\
= & (\lambda n.\lambda s.\lambda z.s(nsz))\bar{0} \\
\to_\beta & \lambda s.\lambda z.s(\bar{0}sz) \\
= & \lambda s.\lambda z.s((\lambda s.\lambda z.z)sz) \\
\to_\beta & \lambda s.\lambda z.s((\lambda z.z)z) \\
\to_\beta & \lambda s.\lambda z.sz \\
= & \bar{1}
\end{array}
$$

Y en general se tiene que $\forall n \in \mathbb{N}\, \mathsf{suc}\,\bar{n} \to^*_\beta \overline{n+1}$

Suma

$$\mathsf{suma} =_{def} \lambda m.\lambda n.n(\mathsf{suc})m$$

Con esta definición la siguiente propiedad es cierta $\forall m, n \in \mathbb{N}(\mathsf{suma}\,\bar{m}\,\bar{n} \to^\star_\beta \overline{m+n}\ )$.

Producto

$$\mathsf{prod} =_{def} \lambda m.\lambda n.m(\mathsf{suma}\ n)\bar{0}$$

Tal que $\forall m, n \in \mathbb{N}(\mathsf{prod}\,\bar{m}\,\bar{n} \to^\star_\beta \overline{m*n}\ )$

Test de cero

$$\mathsf{iszero} =_{def} \lambda m.m(\lambda x.\mathsf{false})\mathsf{true}$$

Y se cumple $\mathsf{iszero}\,\bar{0} \to^\star \mathsf{true}$ $\quad$ $\mathsf{iszero}\,(\overline{n+1}) \to^\star_\beta \mathsf{false}$

4.3 Estructuras de datos

También se pueden definir estructuras de datos en Cálculo Lambda, utilizando abstracciones. algunas de las estructuras mas utilizadas son los pares y listas, las cuales se definen a continuación.

4.3.1 Pares

Los pares son una estructura de datos que almacena dos elementos, son tuplas de dimensión 2. Se representan como una función que toma como argumentos, los dos elementos del par y una función y aplica esta función a los componentes del par.

Constructor de pares

$$\mathsf{pair} =_{def} \lambda f\lambda s\lambda b.bfs$$

Proyección del primer elemento

$$\mathsf{fst} =_{def} \lambda p.p\,\mathsf{true}$$

Proyección del segundo elemento

$$\mathsf{snd} =_{def} \lambda p.p\,\mathsf{false}$$

Veamos algunos ejemplos del funcionamiento de fst y snd.

$$\mathsf{fst}\ (\mathsf{pair\ a\ b}) \to^*_\beta \mathsf{a}$$

En donde a y b son dos expresiones arbitrarias.

$$
\begin{array}{rl}
 & \mathsf{fst}\ (\mathsf{pair}\ \mathsf{a}\ \mathsf{b}) \\
= & \mathsf{fst}\ ((\lambda f.\lambda s.\lambda x.x f s)\ \mathsf{a}\ \mathsf{b}) \\
\to_\beta & \mathsf{fst}\ ((\lambda s.\lambda x.x\ \mathsf{a}\ s)\ \mathsf{b}) \\
\to_\beta & \mathsf{fst}\ (\lambda x.x\ \mathsf{a}\ \mathsf{b}) \\
= & (\lambda p.p\ \mathsf{true})(\lambda x.x\ \mathsf{a}\ \mathsf{b}) \\
\to_\beta & (\lambda x.x\ \mathsf{a}\ \mathsf{b})\ \mathsf{true} \\
\to_\beta & \mathsf{true}\ \mathsf{a}\ \mathsf{b} \\
= & (\lambda x.\lambda y.x)\ \mathsf{a}\ \mathsf{b} \\
\to_\beta & (\lambda y.\mathsf{a})\ \mathsf{b} \\
\to_\beta & \mathsf{a}
\end{array}
$$

Ahora veamos que se cumple

$$\mathsf{snd}\ (\mathsf{pair}\ \mathsf{a}\ \mathsf{b}) \to_\beta^* \mathsf{b}$$

$$
\begin{array}{rl}
 & \mathsf{snd}\ (\mathsf{pair}\ \mathsf{a}\ \mathsf{b}) \\
= & \mathsf{snd}\ ((\lambda f.\lambda s.\lambda x.x f s)\ \mathsf{a}\ \mathsf{b}) \\
\to_\beta & \mathsf{snd}\ ((\lambda s.\lambda x.x\ \mathsf{a}\ s)\ \mathsf{b}) \\
\to_\beta & \mathsf{snd}\ (\lambda x.x\ \mathsf{a}\ \mathsf{b}) \\
= & (\lambda p.p\ \mathsf{false})(\lambda x.x\ \mathsf{a}\ \mathsf{b}) \\
\to_\beta & (\lambda x.x\ \mathsf{a}\ \mathsf{b})\ \mathsf{false} \\
\to_\beta & \mathsf{false}\ \mathsf{a}\ \mathsf{b} \\
= & (\lambda x.\lambda y.y)\ \mathsf{a}\ \mathsf{b} \\
\to_\beta & (\lambda y.y)\ \mathsf{b} \\
\to_\beta & \mathsf{a}
\end{array}
$$

Y de esta forma se puede observar que las funciones fst y snd están bien definidas

4.3.2 Listas

Las listas son la estructura de datos mas utilizada en programación funcional, para definirlas en Cálculo Lambda se usa una representación por pares. Como se ve a continuación.

Las listas se representan como un par. En donde el primer elemento del par es la cabeza y el segundo elemento es la cola. La lista `x : xs` se define como $\langle \mathtt{x}, \mathtt{xs} \rangle$. Sin embargo esta definición no nos da una representación para la lista vacía, para solucionarlo se da una representación de la lista vacía como un término α-equivalente a fst.

Lista vacía

$$\mathsf{nil} =_{def} \mathsf{false}$$

Constructor `cons`

$$\mathsf{cons} =_{def} \mathsf{pair}$$

Función para obtener la cabeza de la lista

$$\mathsf{head} =_{def} \mathsf{fst}$$

Función para obtener la cola de la lista

$$\mathsf{tail} =_{def} \mathsf{snd}$$

Test de la lista vacía

$$\mathsf{isnil} =_{def} \lambda l.l(\lambda h.\lambda t.\lambda d.\mathsf{false})\mathsf{true}$$

Esta representación es usada por lenguajes de programación actuales como *Scheme*.

5 Propiedades de la semántica del Cálculo Lambda

El Cálculo Lambda es considerado como el lenguaje funcional mas pequeño que existe, esto debido a la simpleza de su sintaxis y semántica operacional. En algunos casos este sistema es reconocido como el lenguaje ensamblador de los lenguajes funcionales.

La importancia de este lenguaje nos lleva a preguntarnos como se comporta respecto a las propiedades de la semántica operacional que se estudiaron en la nota anterior, a continuación se hace un análisis de estas propiedades en el caso especifico del Cálculo Lambda.

5.1 No terminación del Cálculo Lambda

En la nota de clase anterior, vimos que el lenguaje EA con asignaciones locales cumplía la propiedad de terminación, es decir, que cualquier expresión correcta del lenguaje puede evaluarse hasta llegar a una expresión bloqueada.

Sin embargo esta propiedad no es válida en el Cálculo Lambda, al existir expresiones que no tienen una forma normal. La no terminación del Cálculo Lambda se debe principalmente al concepto de auto-aplicación, es decir, cuando un lambda termino se aplica a si mismo, como podemos ver en el ejemplo siguiente.

Ejemplo 5.1 (No terminación del Cálculo Lambda)**.** Se presenta un clásico ejemplo de no terminación. Sean $\omega =_{def} \lambda x.xx$ y $\Omega =_{def} \omega\omega$ Utilizando la semántica operacional dada por la regla de β-reducción se genera la siguiente secuencia de evaluación:

$$\Omega = \omega\omega = (\lambda x.xx)\omega \rightarrow_\beta \omega\omega = \Omega$$

Se puede observar que Ω se reduce a sí mismo en un paso de β-reducción, lo que genera la sucesión infinita de reducción

$$\Omega \rightarrow_\beta \Omega \rightarrow_\beta \Omega \rightarrow_\beta \Omega \rightarrow_\beta \Omega \rightarrow_\beta \cdots$$

Por lo que Ω no tiene forma normal, lo cuál representa a un programa cuya evaluación se cicla infinitamente.

5.2 No determinismo

Otra propiedad importante estudiada sobre la semántica de EA es el determinismo de su semántica operacional, esta propiedad indica que cada expresión tiene una única secuencia de evaluación.

Empero la semántica operacional del cálculo lambda no es determinista, esto quiere decir que una misma expresión puede tener dos reductos distintos. Veámoslo con un ejemplo.

Ejemplo 5.2 (No determinismo del Cálculo Lambda)**.** Consideremos el siguiente lambda término:

$$(\lambda x.(\lambda y.yx)z)v$$

Esta expresión tiene dos posibles reductos distintos.

- Si tomamos como redex $(\lambda x.(\lambda y.yx)z)v$ entonces se genera el reducto $(\lambda y.yv)z$.
- Si el redex es $(\lambda x.(\lambda y.yx)z)v$ se obtiene el reducto $(\lambda x.zx)v$.

Lo que define dos posibles secuencias de reducciones. Si continuamos con la evaluación de la expresión, en un paso mas de β-reducción en ambos casos se llega a la expresión zv.

El no determinismo en un sistema representa un problema serio si queremos usarlo como un modelo de cómputo válido, pues se debe garantizar que el resultado final es el mismo sin importar la secuencia de ejecución utilizada.

5.3 Confluencia

El no determinismo del Cálculo Lambda no es un problema gracias a la propiedad de confluencia probada por Church y Rosser, que garantiza que a pesar del no determinismo de la función de β-reducción, la evaluación de una expresión converge en un lambda término.

Teorema 5.1 (Confluencia o propiedad de Church-Rosser)**.** Si $e \to_\beta^* e_1$ y $e \to_\beta^* e_2$ entonces existe un término t tal que $e_1 \to_\beta^* t$ y $e_2 \to_\beta^* t$.

Esta propiedad garantiza que sin importar el camino que se tome la evaluación de una expresión va a coincidir en un mismo término.

Corolario 5.2 (Unicidad de formas normales)**.** Para cualquier expresión e si $e \to_\beta^* e_f$ y $e \to_\beta^* e'_f$ tal que, tanto e_f como e'_f están bloqueadas, entonces $e_f = e'_f$ salvo α-equivalencias. Es decir, la forma normal de una expresión es única.

Este corolario permite utilizar el Cálculo Lambda como un sistema de cómputo válido al garantizar que el resultado final de una expresión será siempre el mismo sin importar el proceso de evaluación.

Sin embargo aún hay un problema presente en el uso de la función de β-reducción como semántica operacional, ya que dependiendo del proceso de evaluación se podría no llegar a la forma normal de la expresión, veamos el siguiente ejemplo.

Ejemplo 5.3. Consideremos la expresión

$$(\lambda x.y)\Omega$$

- Si se toma como redex $(\lambda x.y)\Omega$ se obtiene el reducto y que es la forma normal de la expresión.
- Por el otro lado si tomamos como redex $(\lambda x.y)\Omega$ la expresión diverge por lo que jamás alcanza su forma normal.

El ejemplo anterior muestra que usar el Cálculo Lambda por si sólo como lenguaje de programación puede resultar en expresiones con resultados bien definidos cuya evaluación se cicle infinitamente.

Para corregir esto se deben definir restricciones o estrategias sobre la función de β-reducción.

6 Recursión y combinadores de punto fijo

La recusión es un concepto fundamental cuando se estudia lenguajes de programación, especialmente en el paradigma funcional, ya que es el principal mecanismo mediante e cuál se definen funciones. Por lo que es importante entender como funciona la recursión desde el lado de diseño de estos lenguajes.

El Cálculo Lambda es un lenguaje puramente funcional en el sentido de que todo puede ser definido como una función. Sin embargo hasta ahora no hemos definido ninguna función recursiva.

Intentemos definir una clásica función recursiva sobre los números naturales, la función factorial. Un primer intento podría ser la siguiente expresión:

$$\mathsf{fact} =_{def} \lambda x.\mathsf{if}(\mathsf{iszero}\, x)\, \bar{1}\, (\mathtt{prod}\, x\, (\mathsf{fact}\, (\mathsf{pred}\, x)))$$

Esta definición es un lambda término correcto, ahora veamos que pasa si lo evaluamos con un número en especifico.

$$\begin{array}{rl} & \mathsf{fact}\, \bar{2} \\ =_{def} & (\lambda x.\mathsf{if}(\mathsf{iszero}\, x)\, \bar{1}\, (\mathtt{prod}\, x\, (\mathsf{fact}\, (\mathsf{pred}\, x)))\, \bar{2} \\ \to_{\beta} & \mathsf{if}(\mathsf{iszero}\ \bar{2})\, \bar{1}\, (\mathtt{prod}\, \bar{2}\, (\mathsf{fact}\, (\mathsf{pred}\, \bar{2}))) \\ \to_{\beta}^{*} & \mathsf{if}(\mathsf{false})\, \bar{1}\, (\mathtt{prod}\, \bar{2}\, (\mathsf{fact}\, (\mathsf{pred}\, \bar{2}))) \\ \to_{\beta}^{*} & \mathtt{prod}\, \bar{2}\, (\mathsf{fact}\, (\mathsf{pred}\, \bar{2})) \\ \to_{\beta}^{*} & \mathtt{prod}\, \bar{2}\, (\mathsf{fact}\, \bar{1}) \\ \not\to_{\beta} & \end{array}$$

El problema con esta definición de la función factorial es que en la expresión $\mathtt{prod}\, \bar{2}\, (\mathsf{fact}\, \bar{1})$ la presencia de fact es como una variable libre, por esta razón la expresión se bloquea y entonces la forma normal de $\mathsf{fact}\, \bar{2}$ es $\mathtt{prod}\, \bar{2}\, (\mathsf{fact}\, \bar{1})$ y lo que buscamos es que la forma normal sea $\bar{2}$ pues es el resultado esperado de la ejecución.

Lo que se busca entonces es que fact encuentre su valor como la misma definición de la función que estamos definiendo para poder dar una definición recursiva. Una solución para esto podría ser que la definición de la función tenga un nuevo parámetro que sea la misma función que estamos definiendo, resultando en la siguiente definición:

$$\mathsf{fact}' =_{def} \lambda f.\lambda x.\mathsf{if}(\mathsf{iszero}\, x)\, \bar{1}\, (\mathtt{prod}\, x\, (f\, (\mathsf{pred}\, x)))$$

Y de esta forma ya sólo necesitamos un operador que nos permita aplicar la función fact consigo mismo de forma automatizada, tantas veces como sea necesario. Es decir, una expresión F que cumpla la siguiente propiedad:

$$F\,\mathsf{fact}' \to^*_\beta \mathsf{fact}'\,(F\,\mathsf{fact}')$$

Esta expresión F es lo que se conoce como un combinador de punto fijo.

Definición 6.1 (Combinador de punto fijo)**.** Un lambda término cerrado F es un combinador de punto fijo sí y sólo si cumple alguna de las siguientes condiciones:

1. $F\,g \to^*_\beta g\,(F\,g)$
2. $F\,g \equiv_\beta g\,(F\,g)$, es decir, existe un término t tal que $F\,g \to^*_\beta t$ y $g\,(F\,g) \to^*_\beta t$

En este caso para poder definir la función recursiva fact basta usar un operador de punto fijo. Por lo que la definición queda como sigue.

$$\begin{array}{rcl} \mathsf{fact} & =_{def} & F\,\mathsf{fact}' \\ \mathsf{fact}' & =_{def} & (\lambda f.\lambda x.\mathsf{if}\,(\mathsf{iszero}\,x)\,\bar{1}\,(\mathtt{prod}\,x\,(f\,(\mathsf{pred}\,x))) \end{array}$$

De esta forma, el fact de 2 se calcula de la siguiente forma:

$$\begin{array}{rl} & \mathsf{fact}\,\bar{2} \\ =_{def} & (F\,\mathsf{fact}')\,\bar{2} \\ \to^*_\beta & \mathsf{fact}'\,(F\,\mathsf{fact}')\,\bar{2} \\ =_{def} & (\lambda f.\lambda x.\mathsf{if}\,(\mathsf{iszero}\,x)\,\bar{1}\,(\mathtt{prod}\,x\,(f\,(\mathsf{pred}\,x)))\,(F\,\mathsf{fact}')\,\bar{2} \\ \to_\beta & (\lambda x.\mathsf{if}\,(\mathsf{iszero}\,x)\,\bar{1}\,(\mathtt{prod}\,x\,((F\,\mathsf{fact}')\,(\mathsf{pred}\,x)))\,\bar{2} \\ \to_\beta & \mathsf{if}\,(\mathsf{iszero}\,\bar{2})\,\bar{1}\,(\mathtt{prod}\,\bar{2}\,((F\,\mathsf{fact}')\,(\mathsf{pred}\,\bar{2}))) \\ \to^*_\beta & \mathtt{prod}\,\bar{2}\,((F\,\mathsf{fact}')\,(\mathsf{pred}\,\bar{2})) \\ \to^*_\beta & \mathtt{prod}\,\bar{2}\,((F\,\mathsf{fact}')\,\bar{1}) \\ \to^*_\beta & \mathtt{prod}\,\bar{2}\,(\mathsf{fact}'\,(F\,\mathsf{fact}')\,\bar{1}) \\ \to^*_\beta & \mathtt{prod}\,\bar{2}\,(\mathsf{if}\,(\mathsf{iszero}\,\bar{1})\,\bar{1}\,(\mathtt{prod}\,\bar{1}\,((F\,\mathsf{fact}')\,(\mathsf{pred}\,\bar{1})))) \\ \to^*_\beta & \mathtt{prod}\,\bar{2}\,(\mathtt{prod}\,\bar{1}\,((F\,\mathsf{fact}')\,(\mathsf{pred}\,\bar{1}))) \\ \to^*_\beta & \mathtt{prod}\,\bar{2}\,(\mathtt{prod}\,\bar{1}\,((F\,\mathsf{fact}')\,\bar{0})) \\ \to^*_\beta & \mathtt{prod}\,\bar{2}\,(\mathtt{prod}\,\bar{1}\,(\mathsf{if}\,(\mathsf{iszero}\,\bar{0})\,\bar{1}\,(\mathtt{prod}\,\bar{0}\,((F\,\mathsf{fact}')\,(\mathsf{pred}\,\bar{0}))))) \\ \to^*_\beta & \mathtt{prod}\,\bar{2}\,(\mathtt{prod}\,\bar{1}\,(\mathsf{if}\,(\mathsf{true})\,\bar{1}\,(\mathtt{prod}\,\bar{0}\,((F\,\mathsf{fact}')\,(\mathsf{pred}\,\bar{0}))))) \\ \to^*_\beta & \mathtt{prod}\,\bar{2}\,(\mathtt{prod}\,\bar{1}\,\bar{1}) \\ \to^*_\beta & \mathtt{prod}\,\bar{2}\,\bar{1} \\ \to^*_\beta & \bar{2} \\ \not\to^*_\beta & \end{array}$$

Observación. Es muy importante notar que en este caso fact ' ya no es una variable libre sino la misma definición dada anteriormente, gracias al uso del operador de punto fijo.

Podemos ver que con la existencia de un combinador de punto fijo podemos definir funciones recursivas en el Cálculo Lambda ya que nos permiten aplicar una función consigo misma tantas veces como sea necesario.

Ejemplos (Combinadores de punto fijo)**.** En el Cálculo Lambda existen diferentes combinadores de punto fijo, algunos de ellos se muestran a continuación:

- (**Curry-Rosser**) $\mathsf{Y} =_{def} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

- (**Turing**) $\mathsf{V} =_{def} UU$ en donde $U =_{def} \lambda f.\lambda x.x(ffx)$
- (**Estricto**) $\mathsf{Z} =_{def} \lambda f.(\lambda x.f(\lambda v.xxv))(\lambda x.f(\lambda v.xxv))$
- (**Klop**) $K =_{def} \underbrace{LL\ldots L}_{26\ veces}$ en donde

  $L =_{def} \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$

Proposición 6.2. El combinador Y es un combinador de punto fijo.

Demostración.

$$\begin{array}{rl} & \mathsf{Y}\, g \\ =_{def} & (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\, g \\ \to_\beta & (\lambda x.g(xx))(\lambda x.g(xx)) \\ \to_\beta & g((\lambda x.g(xx))(\lambda x.g(xx))) \end{array}$$

Por otro lado.

$$\begin{array}{rl} & g\,(\mathsf{Y}\, g) \\ =_{def} & g\,((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\, g) \\ \to_\beta & g\,((\lambda x.g(xx))(\lambda x.g(xx))) \end{array}$$

entonces $\mathsf{Y}\, g \equiv_\beta g\,(\mathsf{Y}\, g)$
$\therefore$ Y es un combinador de punto fijo. □

7 El lenguaje ISWIM

El Cálculo Lambda que hemos estudiado hasta ahora es demasiado primitivo para ser considerado un lenguaje de programación, de la misma forma que las máquinas de Turing son demasiado primitivas para ser consideradas una computadora real.

En los años 50 y 60 se descubrió la conexión entre los lenguajes de programación y varios aspectos del Cálculo Lambda, debido al deseo de especificar el significado del lenguaje ALGOL 60 y de formalizar el estudio de los lenguajes de programación empleando sistemas matemáticos.

Uno de los primeros lenguajes de programación formales, basado en el cálculo lambda, es ISWIM acrónimo para la frase *If you See What I Mean*, creado por Peter Landin. Este lenguaje nunca fue implementada directamente, pero sirvío de base para explorar aplicaciones e implementaciones mediante máquinas abstractas.

La definición general de ISWIM es:

$$e ::= x \mid \lambda x.e \mid e\, e \mid c \mid o\,(e, \ldots, e)$$

donde c es una constante primitiva y o es un operador primitivo, tomados de un conjunto de constantes $\mathcal{C}$ y de un conjunto de operadores dados $\mathcal{O}$. Por ejemplo, la instancia de ISWIM relacionada al lenguaje que definimos puramente en Cálculo Lambda consiste de:

$$\mathcal{C} = \{\mathsf{true}, \mathsf{false}, 0, 1, 2\ldots\} \qquad \mathcal{O} = \{\mathsf{suc}, \mathsf{pred}, \mathsf{iszero}, \mathsf{if}, \mathtt{suma}, \mathtt{prod}, \mathsf{not}, \mathsf{and}\}$$

La semántica operacional, dada con la relación de transición $\to$, se define como la unión de la

β-reducción con la llamada δ-reducción, es decir $\rightarrow =_{def} \rightarrow_\beta \cup \rightarrow_\delta$ donde la δ-reducción define el comportamiento de los operadores primitivos. Por ejemplo:

$$\texttt{suma}(m, n) \rightarrow_\delta m + n \qquad \textsf{if}\ (\textsf{false}, e_2, e_3) \rightarrow_\delta e_3$$

La filosofía de este lenguaje de programación es la que siguen algunos lenguaje de programación hoy en día como son Scheme, Lisp y ML.

Referencias

[1] Miranda Perea F., González Huesca L., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM, Semestre 2021-1.

[2] Ramírez Pulido K., Soto Romero M., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM, Semestre 2021-2

[3] Keller G., O'Connor-Davis L., Class Notes from the course Concepts of programming language design, Department of Information and Computing Sciences, Utrecht University, The Netherlands, Fall 2020.

[4] Rojas R., A Tutorial Introduction to the Lambda Calculus, Freie Universitat Berlin, 2015

[5] Barendregt H., Lambda Calculi with Types, Freie, Catholic University Nijmegen.

[6] Harper R., Practical Foundations for Programming Languages. Working draft, 2010.

[7] Mitchell J., Foundations for Programming Languages. MIT Press 1996.

[8] Krishnamurthi S., Programming Languages Application and Interpretation; Version 26.04.2007.