Programación Declarativa, 2021-1
Nota de clase 6: Fundamentos Teóricos de la Programación Funcional*

Karla Ramírez Pulido Manuel Soto Romero Javier Enríquez Mendoza

6 de noviembre de 2020
Facultad de Ciencias UNAM

La Programación Funcional, tiene sus fundamentos principalmente en el Cálculo λ y en la Teoría de Categorías. En esta nota daremos un breve repaso al Cálculo λ, definido en la década de 1930 por Alonzo Church, estudiado en los cursos de Lenguajes de Programación con el fin de entender cómo son aplicados estos conceptos en un lenguaje de Programación Funcional real.

6.1. Sintaxis

El Cálculo λ se compone de una notación para representar expresiones, llamadas *términos* λ. Existen tres tipos de términos: *variables, abstracciones* λ y *aplicaciones de función*. Para construir estos términos, se tiene la siguiente gramática:

```
<expr> ::= <var>
         | (λ<var>.<expr>)
         | (<expr> <expr>)

<var> ::= x | y | z | ...
```

6.1.1. Variables

Las variables son representadas mediante letras minúsculas. Algunos ejemplos de variables son:

$$x \qquad y \qquad z \qquad w$$

6.1.2. Abstracciones λ

Las funciones, abstracciones λ o simplemente abstracciones, representan la definición de funciones *anónimas*, esto es, no tienen un nombre asociado. Tales abstracciones se componen de un encabezado y un cuerpo. El encabezado indica el nombre del parámetro formal de la función y el cuerpo de ésta se compone de las expresiones que debe evaluar la función. Todo lo que se encuentra antes del punto es el encabezado y lo que se encuentra después es el cuerpo. Algunos ejemplos son:

*Basada en las *Notas de clase para el curso de Lenguajes de Programación.*

$$\lambda x.x \qquad \lambda x.\lambda y.xy \qquad \lambda z.\lambda w.u \qquad \lambda u.\lambda v.vvv$$

Las expresiones anteriores, tienen como encabezado λx, λx, λz y λu respectivamente, mientras que x, $\lambda y.xy$, $\lambda w.u$, $\lambda u.\lambda v.vvv$ conforman el cuerpo de las mismas.

La segunda abstracción de los ejemplos anteriores, es equivalente a la función de dos parámetros $\lambda xy.xy$, sin embargo, la especificación de términos λ no permite definir funciones con más de un parámetro. Una forma de lograr definir este tipo de funciones, es hacer que reciban un único parámetro (el primero) y que regresen como resultado una nueva función que recibe el segundo parámetro y así sucesivamente, de forma tal que las aplicaciones a la función se vayan resolviendo parcialmente. Esta técnica recibe el nombre de *currificación*. Veamos algunos ejemplos:

$$\begin{aligned} \lambda xy.xy &\rightarrow \lambda x.\lambda y.xy \\ \lambda zw.u &\rightarrow \lambda z.\lambda w.u \\ \lambda abc.cba &\rightarrow \lambda a.\lambda b.\lambda c.cba \end{aligned}$$

6.1.3. Aplicaciones de función

La regla de producción `(<expr> <expr>)` de la gramática para generar términos λ, representa la aplicación de funciones. Dada una aplicación de función $(e_1\ e_2)$, se dice que e_1 está en posición de función y que e_2 está en la posición de argumento. De esta forma, una aplicación se función representa el proceso de evaluar una función con su respectivo parámetro real. Los paréntesis pueden omitirse. Por ejemplo, la siguiente expresión:

$$\lambda x.x\ \lambda y.y$$

representa la aplicación de la función $\lambda x.x$ con la función $\lambda y.y$. De este ejemplo podemos apreciar que el incluir paréntesis, facilita la lectura o ayudan a entender mejor el orden de evaluación. Algunos ejemplos:

$$(\lambda x.x)\,(\lambda y.y) \qquad xy \qquad (\lambda w.ws)\,(\lambda y.yy)\,u$$

En caso de no haber paréntesis explícitos, la aplicación de funciones asocia a la izquierda.

6.1.4. Alcance

El alcance de una variable, es la región de en la cual éstas alcanzan su valor. En el Cálculo λ el alcance de toda variable se encuentra en el cuerpo de las funciones.

Por ejemplo, en la función $\lambda x.x$ el alcance de la variable x está en el cuerpo. Se dice que la variable x está *ligada* pues se encuentra en el cuerpo de la función que toma como parámetro a la misma variable (*de ligado*). Si una variable en el cuerpo de una función no es precedida por una λ y una variable con el mismo nombre, en el encabezado, se dice que es *libre* en la expresión. Por ejemplo, en la siguiente función:

$$\lambda x.xy$$

La variable x está ligada, mientras que la variable y se encuentra libre. Otro ejemplo se da en:

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

Se observa que en el lado izquierdo de la aplicación $(\lambda x.x)$ la variable x se encuentra ligada, mientras que en el lado derecho $(\lambda y.x)$, x se encuentra libre. Es importante observar que la x de la primera subexpresión es ajena a la de la segunda subexpresión.

Se define formalmente a las variables libres como sigue:

- x está libre en y, con y cualquier variable (incluyendo a x)
- x está libre en $(\lambda y.e)$ si y sólo si $x \neq y$ y x se encuentra libre en e, siendo e una expresión cualquiera
- x está libre en $e_1 e_2$ si y sólo si x está libre en e_1 o en e_2, con e_1 y e_2 expresiones cualquiera

Los identificadores ligados se definen formalmente como sigue:

- x se encuentra ligado en $(\lambda x.e)$
- x se encuentra ligado en $(\lambda y.e)$ con $x \neq y$ si y sólo si x está ligado en e
- x está ligado en $e_1 e_2$ si y sólo si, x está ligado en e_1 o e_2

Se observa, que gracias a la tercera regla de ambas definiciones, éstas no son excluyentes entre sí, es decir, una variable puede ser libre y ligada al mismo tiempo. Una expresión e que no tiene variables libres, se dice que es una expresión *cerrada* o un *combinador*.

6.1.5. α-equivalencias

Los nombres de las variables ligadas, en realidad no son importantes en el contexto del Cálculo λ. Por ejemplo, $\lambda x.x$ y $\lambda y.y$ representan a la misma función y lo único que cambia es el nombre de la variable. A esto se le conoce como relación de α-*equivalencia*. Sin embargo, la relación tiene ciertas restricciones:

$$\lambda V.E \equiv_\alpha \lambda W.E\,[V := W] \text{ si } W \notin \mathit{libres}\,(E)$$

Algunos ejemplos:

$$\begin{aligned} \lambda x.x &\equiv_\alpha \lambda x.x \\ \lambda x.x &\equiv_\alpha \lambda y.y \\ \lambda x.\,(\lambda x.x)\,x &\equiv_\alpha \lambda y.\,(\lambda x.x)\,y \end{aligned}$$

6.2. Semántica

La regla de β-reducción expresa el comportamiento de las aplicaciones de función y ésta es justamente la regla semántica del Cálculo λ, es decir, la forma de evaluar una expresión. Esta regla se define como sigue:

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

Es decir, se deben sustituir todas las apariciones de x por s en e. Lo anterior expresa la idea de una función a la cual se le asigna el parámetro real a su parámetro formal (paso de parámetros). A la expresión $(\lambda x.e)$ se le llama *redex*, mientras que a la expresión $e\,[x := s]$ se le llama *reducto*. Algunos ejemplos:

$$\begin{array}{rcl} (\lambda x.x)\, x & \to_\beta & x \\ (\lambda x.x)\, (\lambda x.x) & \to_\beta & \lambda x.x \\ (\lambda x.x)\, (\lambda y.y) & \to_\beta & \lambda y.y \\ (\lambda x.\lambda y.\, (xy))\, (\lambda x.x) & \to_\beta & \lambda y.\, (\lambda x.x)\, y \quad \to_\beta \quad \lambda y.y \end{array}$$

Dada una expresión e, si no existe e' tal que $e \to_\beta e'$, se dice que e se encuentra en Forma Normal. Es decir, dada una expresión, si ésta no puede reducirse más mediante β-reducciones, se dice entonces que se encuentra en Forma Normal. Por ejemplo, todas las reducciones anteriores, están en Forma Normal.

También existe expresiones, que no tienen una Forma Normal pues sin importar cuantas reducciones se apliquen, siempre se obtiene una nueva expresión que puede ser reducida, en este caso, se dice que la expresión *diverg*e y por lo tanto no tiene Forma Norma. Por ejemplo, sean $\omega =_{def} \lambda x.xx$ y $\Omega =_{def} \omega\omega$, entonces:

$$\Omega =_{def} \omega\omega =_{def} (\lambda x.xx)\, \omega \to_\beta \omega\omega =_{def} \Omega$$

De manera que Ω se reduce a sí misma y por lo tanto diverge:

$$\Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \Omega \to_\beta ...$$

6.3. Expresiones aritméticas

Con el Cálculo λ podemos modelar distintos tipos de datos tales como números, valores booleanos, tuplas, listas, entre muchas otras. Por supuesto, la definición de cada uno de estos tipos de datos hace uso de funciones y su aplicación. En esta sección construiremos dos tipos de datos: expresiones aritméticas y expresiones booleanas.

Para poder usar operaciones aritméticas sobre números naturales, añadiremos una representación ampliamente usada conocida como los *Numerales de Church*. Esta forma de representación incluye una definición para el cero y el resto de naturales se construyen a partir de la función sucesor. De esta forma, podemos representar números con funciones que reciben una función sucesor (s) y una función cero (z), se definen cada uno de estos números aplicando s tantas veces como sea necesario a z.

$$
\begin{array}{rcl}
0 & =_{def} & \lambda s.\lambda z.z \\
1 & =_{def} & \lambda s.\lambda z.sz \\
2 & =_{def} & \lambda s.\lambda z.s\,(sz) \\
 & ... & \\
n & =_{def} & \lambda s.\lambda z.\underbrace{s(...(s}_{n\ veces}\,z)...)
\end{array}
$$

Por supuesto, todo depende de la definición de sucesor que demos. Una definición de esta función podría ser:

$$S =_{def} \lambda n.\lambda a.\lambda b.a\,(nab)$$

Por ejemplo, para calcular el sucesor de cero (1), tenemos:

$$
\begin{array}{rcl}
S0 & =_{def} & (\lambda n.\lambda a.\lambda b.a\,(nab))\,0 \\
 & \rightarrow_{\beta} & \lambda a.\lambda b.a\,(0ab) \\
 & =_{def} & \lambda a.\lambda b.a\,((\lambda s.\lambda z.z)\,ab) \\
 & \rightarrow_{\beta} & \lambda a.\lambda b.a\,((\lambda z.z)\,b) \\
 & \rightarrow_{\beta} & \lambda a.\lambda b.ab \\
 & \equiv_{\alpha} & \lambda s.\lambda z.sz \\
 & =_{def} & 1
\end{array}
$$

6.3.1. Operaciones aritméticas

Una vez definida esta representación, podemos definir algunas operaciones sobre los números naturales.

Suma

Sumar en realidad consiste en aplicar la función sucesor, por ejemplo, si se desean sumar los números $n + m$, se debe aplicar la función sucesor n veces a m. Por ejemplo, para sumar $2 + 3$, tenemos:

$$
\begin{array}{rcl}
2S3 & =_{def} & (\lambda s.\lambda z.s\,(sz))\,S3 \\
 & \rightarrow_{\beta} & (\lambda z.S\,(Sz))\,3 \\
 & \rightarrow_{\beta} & S\,(S3) \\
 & \rightarrow_{\beta} & ... \\
 & \rightarrow_{\beta} & 5
\end{array}
$$

De esta forma, podemos definir la función suma que recibe dos números naturales n y m como sigue:

$$A =_{def} \lambda n.\lambda m.nSm$$

Debido a la conmutatividad de la suma, tenemos también la definición:

$$A' =_{def} \lambda n.\lambda m.mSn$$

Ejercicio 6.1. Demuestra que las funciones A y A' son equivalentes.

Producto

Similarmente, se define la multiplicación como sigue:

$$\lambda m.\lambda n.\lambda a.m\,(na)$$

Ejercicio 6.2. Calcula el producto de 3×3, es decir

$$(\lambda m.\lambda n.\lambda a.x\,(ya))\,33$$

6.4. Expresiones booleanas

Al igual que con los números naturales, es posible representar expresiones booleanas y operaciones entre éstas. En este sentido, las constantes lógicas *verdadero* y *falso* se representan mediante:

$$\begin{array}{lll} T & =_{def} & \lambda x.\lambda y.x \\ F & =_{def} & \lambda x.\lambda y.y \end{array}$$

La función T recibe dos valores para verdadero y falso respectivamente y regresa el primero de estos (verdadero), mientras que la función F, regresa el segundo (falso).

Operaciones con booleanos

Con estas definiciones es posible definir algunas operaciones del Álgebra Booleano, a continuación se muestran las definiciones de las funciones *not*, *or* y *and* junto con su tabla de verdad.

Negación

$$\begin{array}{lll} \neg & =_{def} & \lambda x.xFT \end{array}$$

$$\begin{array}{ll} \neg F & \neg T \\ =_{def}\ (\lambda x.xFT)\,F & =_{def}\ (\lambda x.xFT)\,T \\ \to_{\beta}\ FFT & \to_{\beta}\ TFT \\ =_{def}\ (\lambda x.\lambda y.y)\,FT & =_{def}\ (\lambda x.\lambda y.x)\,FT \\ \to_{\beta}\ (\lambda y.y)\,T & \to_{\beta}\ (\lambda y.F)\,T \\ \to_{\beta}\ T & \to_{\beta}\ F \end{array}$$

Disyunción

$$\begin{array}{lll} \vee & =_{def} & \lambda x.\lambda y.xTy \end{array}$$

$\vee FF$
$=_{def}\ (\lambda x.\lambda y.xTy)\,FF$
$\to_\beta\ (\lambda y.FTy)\,F$
$\to_\beta\ FTF$
$=_{def}\ (\lambda x.\lambda y.y)\,TF$
$\to_\beta\ (\lambda y.y)\,F$
$\to_\beta\ F$

$\vee FT$
$=_{def}\ (\lambda x.\lambda y.xTy)\,FT$
$\to_\beta\ (\lambda y.FTy)\,T$
$\to_\beta\ FTT$
$=_{def}\ (\lambda x.\lambda y.y)\,TT$
$\to_\beta\ (\lambda y.y)\,T$
$\to_\beta\ T$

$\vee TF$
$=_{def}\ (\lambda x.\lambda y.xTy)\,TF$
$\to_\beta\ (\lambda y.TTy)\,F$
$\to_\beta\ TTF$
$=_{def}\ (\lambda x.\lambda y.x)\,TF$
$\to_\beta\ (\lambda y.T)\,F$
$\to_\beta\ T$

$\vee TT$
$=_{def}\ (\lambda x.\lambda y.xTy)\,TT$
$\to_\beta\ (\lambda y.TTy)\,T$
$\to_\beta\ TTT$
$=_{def}\ (\lambda x.\lambda y.x)\,TT$
$\to_\beta\ (\lambda y.T)\,T$
$\to_\beta\ T$

Conjunción

$$\wedge \quad =_{def} \quad \lambda x.\lambda y.xyF$$

Ejercicio 6.3. Desarrolla la tabla de verdad de la conjunción.

Condicional if0

También es posible definir condicionales. Por ejemplo, se define el condicional $if0$. Si se aplica a un número n y éste resulta ser cero, se obtiene T; en caso contrario, se obtiene F. La definición de esta función se muestra a continuación:

$$if0 =_{def} \lambda x.xF\neg F$$

Veamos la reducción de $if00$ e $if01$:

$if00$
$=_{def}\ (\lambda x.xF\neg F)\,0$
$\to_\beta\ 0F\neg F$
$=_{def}\ (\lambda s.\lambda z.z)\,F\neg F$
$\to_\beta\ (\lambda z.z)\,\neg F$
$\to_\beta\ \neg F$
$\to_\beta\ T$

$if01$
$=_{def}\ (\lambda x.xF\neg F)\,1$
$\to_\beta\ 1F\neg F$
$=_{def}\ (\lambda s.\lambda z.sz)\,F\neg F$
$\to_\beta\ (\lambda z.Fz)\,\neg F$
$\to_\beta\ F\neg F$
$=_{def}\ (\lambda x.\lambda y.y)\,\neg F$
$\to_\beta\ (\lambda y.y)\,F$
$\to_\beta\ F$

Ejercicio 6.4. Usando como base la definición del condicional $if0$, define la función if.

6.5. Integrando números y booleanos

Con los naturales y booleanos definidos podemos definir otras funciones de utilidad.

Función predecesor

Para definir esta función construiremos un par de la forma $(n, n-1)$ y tomaremos el segundo elemento del par como resultado. Un par puede representarse mediante la función:

$$\lambda z.zab$$

La z permite realizar operaciones con el par. Por ejemplo, podemos obtener el primer y segundo elemento auxiliandonos de las funciones T (verdadero) y F (falso).

$$(\lambda z.zab)\, T \rightarrow_\beta Tab \rightarrow_\beta a$$

$$(\lambda z.zab)\, F \rightarrow_\beta Fab \rightarrow_\beta b$$

Podemos definir el par $(n+1, n)$ a partir del par $(n, n-1)$, representado por el parámetro p y extrayendo el primer elemento y aplicando la función sucesor. De esta forma definimos la función Φ como sigue:

$$\Phi =_{def} \lambda p.\lambda z.z\,(S\,(pT))\,(pT)$$

De esta forma el predecesor de un número se obtiene aplicando n veces la función Φ al par $\lambda z.00$ y seleccionando el segundo elemento del par usando la función F.

$$P =_{def} \lambda n.(n\Phi\,(\lambda z.z00))F$$

Es importante notar que bajo esta definición, el predecesor de cero es cero, con lo cual se evita la generación de números negativos. Se deja como ejercicio al lector obtener el predecesor de 1.

Operadores relacionales

Una vez definida la función predecesor, podemos definir una función que verifique si un número es mayor o igual que otro. Por ejemplo. para verificar $2 \geq 1$, basta con aplicar la función predecesor 2 veces a 1.

$$2P1 \rightarrow_\beta ... \rightarrow_\beta 0$$

Con la función $if0$ se verifica si esto se cumple. De esta forma, definimos la función $\geq$ como sigue:

$$\geq =_{def} \lambda x.\lambda y.Z\,(xPy)$$

De la misma forma, podemos definir la función igualdad $x = y$. Si se cumple $x \geq y$ y $y \geq x$, por lo tanto $x = y$. De esta forma, definimos la función $=$ que usa la operación $\wedge$ definida anteriormente.

$$==_{def} (\lambda x.\lambda y. \wedge (\geq xy)\,(\geq yx))$$

De forma similar se pueden definir otras operaciones relacionales.

6.6. Recursión

Ya vimos en las secciones anteriores que es posible representar cualquier valor en el Cálculo λ, sin embargo, nos falta estudiar una de las principales características y es el uso de la recursión. Éste mecanismo se logra en los lenguajes de programación mediante llamadas a la misma función mediante su nombre, sin embargo, no existe como tal un nombre de función en el Cálculo λ, es por esto que requerimos del uso de los llamados *Combinadores de punto fijo.*

Sea la siguiente definición para calcular el factorial de un número[1]:

$$FACT =_{def} \lambda n.if\ n = 0\ then\ 1\ else\ n * f\,(n-1)$$

Es importante mencionar que $FACT$ es sólo una forma de nombrar expresiones que hemos usado para facilitar la lectura de las mismas, sin embargo, no existe como tal este concepto en el Cáculo λ y por lo tanto al tratar de usar esta definición, notamos que la variable f se encuentra libre, lo cual representa un problema, pues en tiempo de ejecución no deben existir variable libres.

$$\begin{array}{rcl} FACT\ 3 & =_{def} & (\lambda n.if\ n = 0\ then\ 1\ else\ n * f\,(n-1))\,3 \\ & \rightarrow_{\beta} & if\ 3 = 0\ then\ 1\ else\ 3 * f2 \\ & \rightarrow_{\beta} & 3 * f2 \\ & \rightarrow_{\beta} & ¿? \end{array}$$

La única forma que tenemos de ligar variables en el Cálculo λ es mediante los parámetros de las funciones, por lo tanto, para ligar la variable f, debemos redefinir la función de forma que ésta sea pasada como parámetro.

$$FACT =_{def} \lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ n * f\,(n-1)$$

Sin embargo, al añadir un parámetro adicional, todas las llamadas subsecuentes a f deben modificarse para recibir una función como parámetro. De esta forma se debe autoaplicar f con el fin de garantizar la recursión.

$$FAC =_{def} \lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ n * (f\,f)\,(n-1)$$

Veamos nuevamente la reducción de factorial de 3, recordar que esta nueva definición debe recibir una función como parámetro, este caso la propia función $FACT$.

[1] Se abusa de la notación para representar la estructura if

$$
\begin{array}{rcl}
(FACT\ FACT)\ 3 & =_{def} & ((\lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ n * (f f)\,(n-1))\ FACT)\,3 \\
& \rightarrow_{\beta} & (\lambda n.if\ n = 0\ then\ 1\ else\ n * (FACT\ FACT)\,(n-1))\,3 \\
& \rightarrow_{\beta} & if\ 3 = 0\ then\ 1\ else\ 3 * (FACT\ FACT)\,(3-1) \\
& \rightarrow_{\beta} & 3 * (FACT\ FACT)\,(3-1) \\
& =_{def} & 3 * ((\lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ n * (f f)\,(n-1))\ FACT)\,2 \\
& \rightarrow_{\beta} & 3 * (\lambda n.if\ n = 0\ then\ 1\ else\ n * (FACT\ FACT)\,(n-1))\,2 \\
& \rightarrow_{\beta} & 3 * if\ 2 = 0\ then\ 1\ else\ 2 * (FACT\ FACT)\,(2-1) \\
& \rightarrow_{\beta} & 3 * 2 * (FACT\ FACT)\,(2-1) \\
& =_{def} & 3 * 2 * ((\lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ n * (f f)\,(n-1))\ FACT)\,1 \\
& \rightarrow_{\beta} & 3 * 2 * (\lambda n.if\ n = 0\ then\ 1\ else\ n * (FACT\ FACT)\,(n-1))\,1 \\
& \rightarrow_{\beta} & 3 * 2 * if\ 1 = 0\ then\ 1\ else\ 1 * (FACT\ FACT)\,(1-1) \\
& \rightarrow_{\beta} & 3 * 2 * 1 * (FACT\ FACT)\,(1-1) \\
& =_{def} & 3 * 2 * 1 * ((\lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ n * (f f)\,(n-1))\ FACT)\,0 \\
& \rightarrow_{\beta} & 3 * 2 * 1 * (\lambda n.if\ n = 0\ then\ 1\ else\ n * (FACT\ FACT)\,(n-1))\,0 \\
& \rightarrow_{\beta} & 3 * 2 * 1 * if\ 0 = 0\ then\ 1\ else\ 0 * (FACT\ FACT)\,(0-1) \\
& \rightarrow_{\beta} & 3 * 2 * 1 * 1 \\
& \rightarrow_{\beta} & 6
\end{array}
$$

Con estos pasos es posible definir recursivamente prácticamente cualquier función, sin embargo, es una manera bastante tediosa y si no se aplica correctamente es propensa a errores. Existen una conjunciones de funciones especiales que realizan esta conversión de forma automática llamados Combinadores de Punto Fijo.

A continuación se definen los conceptos necesarios para entender estas funciones.

6.7. Combinadores

Un combinador es una expresión cerrada, es decir, que no tiene variables libres, éstos permiten la definición de nuevas funciones. Existen una infinidad de combinadores dentro del Cálculo λ, por ejemplo: [5]

- La identidad es el combinador más simple y uno de los más utilizados. Se le conoce como I y se expresa de la siguiente manera:

$$I =_{def} \lambda x.x$$

- La función constante que sin importar el parámetro que reciba, siempre regresa x. Se utiliza en la definición de funciones constantes y se le conoce como K.

$$K =_{def} \lambda x.\lambda y.x$$

- Una versión generalizada de la aplicación de funciones se da con el combinador S.

$$S =_{def} \lambda x.\lambda y.\lambda z.\,(x\ z\ \ (y\ z))$$

Algo interesante de estos dos últimos combinadores, es que existe una completud entre ellos, esto quiere decir que es posible producir cualquier otro combinador a partir de éstos. En otras palabras, los combinadores K y S son una base para el conjunto de combinadores del Cálculo λ. Por ejemplo, definamos el combinador I a partir de S y K.

$$\begin{array}{rcl} Ix & =_{def} & (SKK)\,x \\ & \rightarrow_{\beta} & SKKx \\ & \rightarrow_{\beta} & Kx\,(Kx) \\ & \rightarrow_{\beta} & x \end{array}$$

6.8. El Combinador Y

En matemáticas, se dice que x es un *punto fijo* de una función f si $x = f\,(x)$. En el Cálculo λ existen una serie de combinadores que reciben como parámetro una función y devuelven como resultado el punto fijo de la misma. A estos combinadores se les da el nombre de *combinadores de punto fijo*. [5]

Si F es un combinador de punto fijo y g es una función cualquiera, entonces (Fg) da como resultado el punto fijo de g, es decir: [2]

$$Fg = g\,(Fg)$$

La aplicación de un combinador de punto fijo, permite implementar recursión encapsulando el concepto de la autoaplicación, por ejemplo, supongamos definido un combinador de punto fijo F, aplicando éste a nuestra definición del factorial, tenemos:

$$\begin{array}{rcl} (F\ FACT)\ 3 & =_{def} & (FACT\ (F\ FACT))3 \\ & =_{def} & ((\lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ n * f\,(n-1))\ (F\ FACT))\ 3) \\ & \rightarrow_{\beta} & (\lambda n.if\ n = 0\ then\ 1\ else\ n * (F\ FACT)\,(n-1))\ 3 \\ & \rightarrow_{\beta} & if\ 3 = 0\ then\ 1\ else\ 3 * (F\ FACT)\ 2 \\ & \rightarrow_{\beta} & 3 * (F\ FACT)\ 2 \\ & =_{def} & 3 * (FACT\ (F\ FACT))2 \\ & =_{def} & 3 * ((\lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ n * f\,(n-1))\ (F\ FACT))\ 2) \\ & \rightarrow_{\beta} & 3 * (\lambda n.if\ n = 0\ then\ 1\ else\ n * (F\ FACT)\,(n-1))\ 2 \\ & \rightarrow_{\beta} & 3 * if\ 2 = 0\ then\ 1\ else\ 2 * (F\ FACT)\ 1 \\ & \rightarrow_{\beta} & 3 * 2 * (F\ FACT)\ 1 \end{array}$$

$$
\begin{aligned}
=_{def} \quad & 3 * 2 * (FACT\ (F\ FACT))1 \\
=_{def} \quad & 3 * 2((\lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ n * f\,(n-1))\ (F\ FACT))\ 1) \\
\rightarrow_{\beta} \quad & 3 * 2 * (\lambda n.if\ n = 0\ then\ 1\ else\ n * (F\ FACT)\,(n-1))\ 1 \\
\rightarrow_{\beta} \quad & 3 * 2 * if\ 1 = 0\ then\ 1\ else\ 1 * (F\ FACT)\ 0 \\
\rightarrow_{\beta} \quad & 3 * 2 * 1 * (F\ FACT)\ 0 \\
=_{def} \quad & 3 * 2 * 1 * (FACT\ (F\ FACT))0 \\
=_{def} \quad & 3 * 2 * 1 * ((\lambda f.\lambda n.if\ n = 0\ then\ 1\ else\ n * f\,(n-1))\ (F\ FACT))\ 0) \\
\rightarrow_{\beta} \quad & 3 * 2 * 1 * (\lambda n.if\ n = 0\ then\ 1\ else\ n * (F\ FACT)\,(n-1))\ 2 \\
\rightarrow_{\beta} \quad & 3 * 2 * 1 * if\ 0 = 0\ then\ 1\ else\ 0 * (F\ FACT)\ -1 \\
\rightarrow_{\beta} \quad & 3 * 2 * 1 * 1 \\
\rightarrow_{\beta} \quad & 6
\end{aligned}
$$

Existen distintos combinadores de punto fijo, sin embargo, el más conocido, descubierto por Haskell Curry, es el *Combinador Y*.

Definición 6.1. *Se define al Combinador de punto fijo Y como sigue:*

$$Y =_{def} \lambda f.\,(\lambda x.f\,(xx))\,(\lambda x.f\,(xx))$$

Teorema 6.1. *Y es un combinador de punto fijo.*

Demostración. Debemos probar que Y satisface la propiedad $Yg = g\,(Yg)$ para toda función g del Cálculo λ.

$$
\begin{aligned}
Yg \quad =_{def} \quad & (\lambda f.\,(\lambda x.f\,(xx))\,(\lambda x.f\,(xx)))\,g \\
\rightarrow_{\beta} \quad & (\lambda x.g\,(xx))\,(\lambda x.g\,(xx)) \\
\rightarrow_{\beta} \quad & g\,((\lambda x.g\,(xx))\,(\lambda x.g\,(xx))) \\
\rightarrow_{\beta} \quad & g\,(Yg)
\end{aligned}
$$

□

Ejercicio 6.5. Prueba que en efecto $(Y\ FACT)\,3$ da como resultado 6.

Referencias

[1] Karla Ramírez, Manuel Soto, et. al., *Notas de clase de Lenguajes de Programación*, Facultad de Ciencias UNAM, Revisión 2021-1.

[2] A Tutorial Introduction to the Lambda Calculus. Raúl Rojas. Web. 22 julio 2019 `https://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf`

[3] Programming Languages and Lambda Calculi, Matthias Felleisen, Matthew Flatt, 2003.

[4] Favio E. Miranda, Lourdes del C. González, *Notas de Clase de Lenguajes de Programación,* Facultad de Ciencias UNAM. Revisión 2019-1.

[5] Diego Murillo, *Una implementación Polimórfica de ISWIM,* Tesis de Licenciatura, Facultad de Ciencias UNAM, 2017.