Lenguajes de Programación, 2019-I[*]
Nota de clase 6: Introducción al Paradigma Funcional, Cálculo Lambda sin Tipos[**]

Favio E. Miranda Perea Lourdes Del Carmen González Huesca
Facultad de Ciencias UNAM

11 de septiembre de 2018

En el lenguaje de expresiones aritméticas estudiado anteriormente es posible expresar, por ejemplo, para cada expresión dada e, el cómputo de su doble mediante la operación de suma $e + e$. Sin embargo no es posible expresar el mecanismo de duplicación de una expresión numérica en general, es decir, no es posible escribir un programa que reciba una expresión numérica cualquiera e y devuelva $e + e$. Para esto necesitamos introducir un mecanismo de definición de funciones que capture patrones de cómputo generales que puedan instanciarse para obtener resultados específicos. Para pasar de instancias particulares de $e + e$ a una expresión general reemplazamos las presencias de e por una variable x la cual se marca como sujeta a variación mediante un mecanismo de abstracción y ligado. En esta nota extendemos el lenguaje con este mecanismo de definición conocido como abstracción funcional o abstracción lambda (λ). Esta clase de abstracción es común a diversos lenguajes de programación funcional, en particular en la familia de lenguajes descendientes de LISP la duplicación puede escribirse como (`lambda x (+ x x)`). En general la sintaxis de las expresiones lambda, en SCHEME por ejemplo,es

```
(lambda (x) e)
```

donde `lambda` es una palabra reservada que indica la definición de una función con parámetro x y cuerpo e. La expresión puede leerse como "la función que, para cada x, devuelve e". El parámetro se liga mediante la abstracción lambda y consecuentemente puede ser renombrado de acuerdo a las reglas de la α-equivalencia. Veamos un par de ejemplos mas:

- La función $x \mapsto x^2$ se denota como (`lambda (x) (* x x)`)
- La función $y \mapsto (3 + y) * (y - 4)$ es (`lambda (y) (* (+ 3 y) (- y 4))`)
- También podemos definir funciones con más de un parámetro, por ejemplo la función producto $(x, y) \mapsto x * y$ se define como (`lambda (x y) (* x y)`)

En complemento a la definición de funciones se requiere un mecanismo de aplicación de una función f a un argumento e. Esta operación no requiere de un operador en la sintaxis concreta y se denota simplemente con la escritura en secuencia de ambas expresiones, es decir, la expresión `f y`

[*]Agradecemos la colaboración de Susana Hahn Martín-Lunas en la revisión 2018-I
[**]Estas notas se basan en los libros de Harper y Pierce y en material de Frank Pfenning, Ralf Hinze y Gerwin Klein.

indica que la función f se aplica al argumento y. Por ejemplo la expresión (`lambda` (`z`) (+ `2` `z`)) `3` se refiere a la aplicación de la función (`lambda` (`z`) (+ `2` `z`)) al argumento 3 lo cual debe evaluarse a 5.

El lenguaje formal para estudiar las operaciones de abstracción y aplicación funcional se conoce como cálculo lambda. Este formalismo puede considerarse como el cimiento del paradigma de programación funcional y es parte esencial de los lenguajes funcionales modernos. Aunque es importante observar que no tuvo mayor impacto en el desarrollo de LISP más allá del uso de la notación lambda. Al respecto el mismo McCarthy[1] dice "*Para usar funciones como argumentos, uno necesita una notación para funciones, y parecía natural usar la notación λ de Church (1941). Yo no entendí el resto de su libro de manera que no intenté implementar su mecanismo general para definir funciones*". Esto muestra la humildad de un gran científico, LISP tuvo muchas innovaciones tanto teóricas como prácticas, por ejemplo:

- El uso de recolección de basura[2] como un método para reutilizar celdas de memoria.
- El uso de cerraduras para implementar la resolución estática del alcance.
- La invención de la expresión condicional y su uso para escribir funciones recursivas.
- El uso de funciones de orden superior, es decir, funciones que reciben funciones como argumento y/o devuelven funciones como resultado.

A continuación abandonamos la notación lambda de LISP para estudiar brevemente el cálculo lambda puro y con tipos simples.

1. El Cálculo Lambda Puro λ^U

El cálculo lambda es un formalismo matemático inventado durante finales de la decada de l930 por Alonzo Church, Stephen Kleene y Haskell Curry (lógica combinatoria) con el propósito de fundamentar la matemática. Si bien tal propósito no se cumplió, el mismo Church dijo alguna vez que *el sistema podría tener otras aplicaciones distintas a su uso como una lógica.* Palabras proféticas dado que el cálculo lambda, diseñado para investigar la definición y aplicación de funciones así como diversos principios de recursión, se ha convertido en el cálculo núcleo para estudiar fundamentos de lenguajes de programación. De hecho puede considerarse como un lenguaje ensamblador para el paradigma de programación funcional.

La definición del cálculo lambda puro, denotado de ahora en adelante λ^U, es sorprendentemente simple, he aquí su sintaxis concreta:

$$e ::= x \mid \lambda x.e \mid e\, e$$

Se tiene un conjunto infinito de variables y dos operaciones, la abstracción lambda $\lambda x.e$ declarada en LISP y sus sucesores como `(lambda (x) e)` y la aplicación denotada $e_1\, e_2$.
Es muy importante observar que en el cálculo lambda puro no hay distinción formal entre funciones y argumentos, toda expresión denota a una función, este es el caso extremo de un lenguaje de programación funcional.

[1] John McCarthy. History of LISP. In Richard L. Wexelblat, editor, History of Programming Languages: Proceedings of the ACM SIGPLAN Conference, pages 173-197. Academic Press, June 1-3 1978.

[2] En inglés *Garbage collection.*

1.1. Aplicación

Una expresión de la forma $e_1\, e_2$ se llama *aplicación*. Aunque ambas expresiones son funciones, se dice que la expresión e_1 está en posición de función mientras que la expresión e_2 está en posición de argumento. La idea es que $e_1\, e_2$ denota a la aplicación de la función e_1 al argumento e_2.
La aplicación se asocia a la **izquierda** de manera que $e_1\, e_2\, e_3$ significa $(e_1\, e_2)\, e_3$ y NO $e_1\, (e_2\, e_3)$. Obsérvese que no hay un operador explícito para la aplicación, ésta se denota simplemente mediante la escritura secuencial de dos expresiones. Esto genera un reto importante en el proceso de análisis sintáctico.

1.2. Abstracción lambda

Una expresión de la forma $\lambda x.e$ se llama *abstracción lambda* o simplemente *abstracción*. La variable x es la variable de la abstracción y la expresión e se llama el cuerpo de la abstracción. La idea aquí es que $\lambda x.e$ indica una abstracción de los valores particulares de la variable x en la expresión e lo cual causa que e se considere una función de x, es decir, la expresión $\lambda x.e$ define anónimamente a la función la función $x \mapsto e$ que asocia a cada valor x la expresión e . El punto en una abstracción es un caso particular de la sintaxis de orden superior y denota el ligado de x en e, además indica que el alcance del ligado para x se extiende a la derecha tanto como sea posible. Por ejemplo, $\lambda x.xy$ significa $\lambda x.(xy)$ y no $(\lambda x.x)y$. Para facilitar la escritura de abstracciones usaremos la siguiente convención

$$\lambda x_1 x_2 \ldots x_n.e =_{def} \lambda x_1.\lambda x_2.\ldots.\lambda x_n.e$$

Veamos unos ejemplos:

$$\begin{array}{rcl} \lambda xyz.xz(yz) & = & \lambda x.\lambda y.\lambda z.xz(yz) \\ & = & \lambda x.\lambda y.\lambda z.(xz)(yz) \\ \\ (\lambda xyz.xz)yz & = & \big((\lambda xyz.xz)y\big)z \\ & = & \big((\lambda x.\lambda y.\lambda z.xz)y\big)z \end{array}$$

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`. En particular se tiene que el conjunto de variables libres de una abstracción se define como $FV(\lambda x.e) = FV(e) \setminus \{x\}$. Por ejemplo en $(\lambda xy.xzy)(\lambda z.zy)u$ las dos presencias de x son ligadas así como las dos primeras presencias de y y las dos últimas presencias de z mientras que el resto de las presencias de cada variable son libres. Una expresión e tal que si $FV(e) = \varnothing$ es una expresión *cerrada* también conocida como *combinador*[3].

Ejemplo 1.1 *Algunos términos del cálculo λ y su significado intuitivo son:*

- *x, la función indeterminada x*
- *xy, la aplicación de la función x a la función y, ambas indeterminadas.*
- *xyz, la aplicación de la función xy a z, todas indeterminadas.*

[3]El estudio de los combinadores originó la lógica combinatoria introducida por Haskell Curry. Ejemplos de combinadores son: $I = \lambda x.x$ el combinador más simple, el cual representa la función identidad, $S = \lambda xyz.xz(yz)$ y $K = \lambda x\lambda y.x$

- *$x(yz)$, la aplicación de x a la aplicación de y a z.*
- *$\lambda x.x$ la función que toma un argumento x y devuelve el mismo elemento x, es decir, la función identidad.*
- *$\lambda z.y$, la función que toma un argumento z y devuelve y, es decir, la función constante y.*
- *$\lambda x.\lambda z.x$, la función que para cada x, devuelve la función que para cada z devuelve x, es decir, la función que para cada x devuelve la función constante que devuelve x.*
- *$\lambda x.\lambda y.\lambda z.xyz$, la función que le aplica a cada z el resultado de aplicarle x a y.*
- *$\lambda x.\lambda y.\lambda z.x(yz)$, la función composición de x y y*
- *$\lambda x.xx$, la función que para cada x devuelve la aplicación de x a x.*
- *$(xy)(zw)$, la función que aplica el resultado de aplicar x a y, al resultado de aplicar z a w, todas indeterminadas.*

2. Semántica Operacional

La semántica operacional del cálculo lambda es muy simple y está dada por la cerradura bajo todos los constructores de término de la siguiente regla conocida históricamente como reducción β.

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

Debido a esta regla, a cualquier expresión de la forma $(\lambda x.t)\, s$ se les 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 substitución 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 s.p.g. $y \neq x$ y $y \notin FV(r)$.

De manera más detallada definimos la relación de reducción o evaluación del cálculo lambda $\to_\beta$ como sigue:

- $e \to_\beta e'$ syss en e existe un redex $r =_{def} (\lambda x.t)s$ y e' se obtuvo a partir de e sustituyendo r en e por su reducto $r' =_{def} t[x := s]$. Algunos ejemplos son:

  $(\lambda x.x(xy))r \to_\beta r(ry)$
  $(\lambda x.y)r \to_\beta y$
  $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z$
  $(\lambda y.yv)z \to_\beta zv$

- Dada e si no existe e' tal que $e \to e'$ entonces decimos que e está en forma normal.

- Si $e \to_\beta^\star e_f$ y e_f está en forma normal entonces decimos que e_f es la forma normal de e. Algunos ejemplos son:

  Las expresiones $x, xy, \lambda x.xz$ están todas en forma normal

  La expresión $(\lambda x.y)(\lambda x.x)$ tiene como forma normal a y.

  La expresión $(\lambda x.(\lambda y.yx)z)v$ tiene como forma normal a zv.

2.1. No terminación del cálculo lambda puro

La propiedad de terminación no es válida en el cálculo lambda puro, al existir expresiones e que no cuentan con una forma normal. El ejemplo clásico es sl siguiente: sean $\omega =_{def} \lambda x.xx$ y $\Omega =_{def} \omega\omega$. La semántica operacional nos lleva a la siguiente secuencia de reducción:

$$\Omega = \omega\omega = (\lambda x.xx)\omega \to_\beta (xx)[x := \omega] = \omega\omega = \Omega$$

De manera que Ω se reduce a sí mismo en un paso lo cual genera la sucesión infinita de reducción:

$$\Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \ldots$$

Es decir Ω no tiene una forma normal y representa a un programa cuya evaluación se cicla indefinidamente.

Obsérvese que una razón para la no terminación de expresiones como Ω es el uso de la auto-aplicación, ω es una función que recibe una función x y devuelve xx que es el resultado de aplicar x a sí misma. Por otra parte Ω es la función que resulta al aplicar ω a sí misma. El cálculo lambda puro no prohibe que una expresión se aplique a sí misma lo cual a primera vista parece extraño, pero resulta útil para definir funciones recursivas como veremos más adelante.

2.2. Confluencia

El cálculo lambda tiene una naturaleza no determinista, por ejemplo la expresión $(\lambda x.(\lambda y.yx)z)v$ tiene dos reductos distintos:

$(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z$ al reducir el redex $(\lambda x.(\lambda y.yx)z)v$

$(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v$ al reducir el redex $(\lambda y.yx)z$

En este caso ambos caminos llevan a la misma forma normal, a saber zv, pero debemos preguntarnos si esto siempre sucede. Si queremos usar el cálculo lambda como un sistema de cómputo, el resultado final debe ser el mismo independientemente del camino utilizado. Si esta propiedad, llamada confluencia, fuera falsa para la β-reducción, cualquier intento de usar al cálculo lambda como un lenguaje de programación sería fallido.

Teorema 1 (Confluencia o propiedad de Church-Rosser) *Si $e \to_\beta^\star r$ y $e \to_\beta^\star s$ entonces existe un término t tal que $r \to_\beta^\star t$ y $s \to_\beta^\star t$*

Demostración. La prueba es complicada y se omite. Podría esperarse una prueba sencilla a partir de la prueba de la misma propiedad para la relación $\rightarrow_\beta$. Sin embargo, para este caso la propiedad es falsa. Por ejemplo si $e =_{def} (\lambda y.uyy)(Iz)$ donde $I = \lambda x.x$ entonces $e \rightarrow_\beta u(Iz)(Iz)$, $e \rightarrow_\beta (\lambda y.uyy)z$ y $(\lambda y.uyy)z \rightarrow_\beta uzz$ y uzz está en forma normal pero es imposible reducir $u(Iz)(Iz)$ a uzz en un sólo paso.

Corolario 1 (Unicidad de las formas normales) *Si e tiene una forma normal entonces es única. Es decir si $e \rightarrow^\star e_f$ y $e \rightarrow^\star e'_f$ entonces $e_f = e'_f$ (salvo α-equivalencia).*

Este corolario garantiza que el cálculo lambda puede usarse como un sistema de cómputo. Sin embargo el uso de la reducción β sin restricciones o estrategias podría prohibir la obtención de una forma normal. Por ejemplo considérese el término $e =_{def} (\lambda x.y)\Omega$. Entonces

- $e \rightarrow_\beta y$ y y es la forma normal de e.
- $e \rightarrow_\beta e$ puesto que $\Omega \rightarrow_\beta \Omega$ por lo que por este camino e no alcanzará jamás su forma normal.

Por lo anterior, si se usara directamente el cálculo lambda como lenguaje de programación habría algunos programas que, teniendo un valor bien definido, podrían ciclarse infinitamente.

En las siguientes secciones omitiremos el subindice β en la notación de las reducciones beta $\rightarrow_\beta$, para facilitar la escritura.

3. Representación de booleanos, pares y números naturales

El cálculo lambda puro es lo suficientemente poderoso para representar tipos de datos como booleanos y números naturales, así como estructuras de datos como pares. Veamos algunos ejemplos de importancia, la deducción de estas definiciones se discutio ampliamente en clase.

3.1. Booleanos

- $\mathsf{true} =_{def} \lambda x\lambda y.x$
- $\mathsf{false} =_{def} \lambda x\lambda y.y$
- $\mathsf{ift} =_{def} \lambda v.\lambda t\lambda f.vtf$
- $\mathsf{not} =_{def} \lambda z.z\ \mathsf{false}\ \mathsf{true}$
- $\mathsf{and} =_{def} \lambda x.\lambda y.xy\ \mathsf{false}$

Con estas definiciones es facil cerciorarse de lo siguiente:

- $\mathsf{ift}\ \mathsf{true}\ e_1\, e_2 \rightarrow^\star e_1$
- $\mathsf{ift}\ \mathsf{false}\ e_1\, e_2 \rightarrow^\star e_2$

por lo que el término ift se comporta como el condicional if_then_else.

- $\mathsf{not\ true} \to^\star \mathsf{false}$
- $\mathsf{not\ false} \to^\star \mathsf{true}$
- $\mathsf{and\ false}\ b \to^\star \mathsf{false}$
- $\mathsf{and\ true}\ b \to^\star b$

3.2. Pares

- $\mathsf{pair} =_{def} \lambda f \lambda s \lambda b.bfs$
- $\mathsf{fst} =_{def} \lambda p.p\ \mathsf{true}$
- $\mathsf{snd} =_{def} \lambda p.p\ \mathsf{false}$

Estos términos cumplen lo siguiente:

- $\mathsf{fst}\,(\mathsf{pair}\,s\,t) \to^\star s$
- $\mathsf{snd}\,(\mathsf{pair}\,s\,t) \to^\star t$

3.3. Naturales (numerales de Church)

Los números naturales se implementan como sigue y se conocen como numerales de Church.

- $\bar{0} =_{def} \lambda s.\lambda z.z$
- $\bar{1} =_{def} \lambda s.\lambda z.sz$
- $\bar{2} =_{def} \lambda s.\lambda z.s(sz)$
- $\bar{3} =_{def} \lambda s.\lambda z.s(s(sz))$
- $\bar{n} =_{def} \lambda s.\lambda z.\underbrace{s(\ldots(s}_{n\ veces}\,z)\ldots)$

Con esta definición podemos utilizar los numerales de Church como iteradores al aplicarlos a una función g y un valor inicial i. De esta forma $\bar{n}gi \to^\star \underbrace{g(\ldots(g}_{n\ veces}\,i)\ldots)$

Las operaciones usuales se implementan como sigue:

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

$$\forall n \in \mathbb{N}\big(\mathsf{suc}\,\bar{n} \to^\star \overline{n+1}\,\big)$$

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

$$\forall m, n \in \mathbb{N}\big(\mathsf{suma}\,\bar{m}\,\bar{n} \to^\star \overline{m+n}\,\big)$$

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

$$\forall m, n \in \mathbb{N}\big(\mathsf{prod}\,\bar{m}\,\bar{n} \to^{\star} \overline{m * n}\ \big)$$

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

$$\mathsf{iszero}\ \bar{0} \to^{\star} \mathsf{true} \qquad \mathsf{iszero}\ (\overline{n+1}) \to^{\star} \mathsf{false}$$

- $\mathsf{pred} =_{def} \lambda m.\mathsf{fst}\,(m\,\mathsf{ss}\,\mathsf{zz})$ donde $\mathsf{zz} =_{def} \mathsf{pair}\,\bar{0}\,\bar{0}$ y $\mathsf{ss} =_{def} \lambda p.\mathsf{pair}\,(\mathsf{snd}\,p)(\mathsf{suc}\,(\mathsf{snd}\,p))$. Así se cumple que

$$\mathsf{pred}\,\bar{0} \to^{\star} \bar{0} \qquad \mathsf{pred}\,\overline{n+1} \to^{\star} \bar{n}$$

4. Azúcar Sintáctica

El uso de expresiones λ si bien proporciona una notación lineal simple para definir funciones puede resultar dificil de manejar para el programador. Para evitar su uso directo podemos introducir una definición que se asemeje a la declaración de funciones en lenguajes de programación actuales. Por ejemplo podemos agregar a la sintaxis concreta una declaración de la forma $\mathsf{fun}(x : \mathsf{T}) \Rightarrow e$ para denotar a la función $x \mapsto e$. Es decir, el uso de la declaración fun es simplemente una abreviatura:

$$\mathsf{fun}(x) \Rightarrow e =_{def} \lambda x.e$$

Este proceso de definición se conoce como *azúcar sintáctica*[4] término introducido por Peter Landin, el término se refiere a la extensión de la sintaxis concreta de un lenguaje sin afectar su sintaxis abstracta, propiedades o funcionalidad. El nombre se refiere a que el uso de este tipo de definiciones hace la vida más dulce al programador. La introducción de azúcar sintáctica le da al programador o diseñador una manera alternativa y más práctica para codificar lo cual lleva a un estilo más limpio y claro de programación el cual es más simple de entender. Por otra parte, su uso no debe afectar la expresividad del formalismo ni permitir que el lenguaje haga algo nuevo.

Veamos algunos ejemplos de expresiones endulzadas y su significado real:

- $\mathsf{fun}(x) \Rightarrow 7$ es la función constante 7, $\lambda x.7$
- $\mathsf{fun}(x) \Rightarrow (\mathsf{fun}(y) \Rightarrow x)$ es la función que toma un natural x y devuelve la función constante $y \mapsto x$, es decir, $\lambda x.\lambda y.x$
- $\mathsf{fun}(x) \Rightarrow \mathsf{if}\ x\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true}$ es la función negación $\neg x$, $\lambda x.\mathsf{if}\ x\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true}$
- $\mathsf{fun}(x) \Rightarrow \big(\mathsf{fun}(y) \Rightarrow \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{if}\ y\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false})\ \mathsf{else}\ \mathsf{false}\big)$ es la conjunción $x \wedge y$.
- La composición de funciones $\lambda f.\lambda g.\lambda x.f(gx)$ se puede declarar como

$$\mathsf{fun}(f) \Rightarrow \big(\mathsf{fun}(g) \Rightarrow (\mathsf{fun}(x) \Rightarrow f(gx))\big)$$

[4] syntax sugar

Obsérvese que es complicado escribir funciones con la notación fun porque en el cálculo lambda todas las funciones reciben un sólo parámetro. Esto puede mejorarse al introducir más azúcar sintáctica:

$$\begin{aligned}\mathsf{fun}(x_1,\ldots,x_n)\ \Rightarrow\ e\ \ &=_{def}\\ &\mathsf{fun}(x_1)\ \Rightarrow\ \big(\mathsf{fun}(x_2)\ \Rightarrow\ \ldots(\mathsf{fun}(x_n)\ \Rightarrow\ e)\ldots\big)\\ &=_{def}\ \lambda x_1.\lambda x_2.\ldots.\lambda x_n.e\end{aligned}$$

De esta manera la composición de funciones queda definida como

$$\mathsf{fun}(f,\ g,\ x)\ \Rightarrow\ f(g\,x)$$

Notemos que también podemos introducir la expresión `let` como azúcar sintáctica de la siguiente manera.

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} =_{def}\ (\lambda x.e_2)e_1$$

5. La importancia de nombrar funciones

Hasta ahora tenemos disponible un constructor fun para definir funciones anónimas pero para acercarnos a un lenguaje de programación real nos gustaría poder nombrar funciones mediante declaraciones del tipo

$$\mathsf{fun}\ factorial\,(n)\ \Rightarrow\ e$$

Es importante mencionar que el nombrar funciones no es simplemente una característica que facilita el entendimiento de un programa, sino que permite implementar funciones mediante recursión lo cual no es posible con nuestra declaración fun para funciones anónimas. Considérese el siguiente ejemplo:

$$\begin{aligned}2^0\ &=\ 1\\ 2^{n+1}\ &=\ 2\times 2^n,\ \ n>0\end{aligned}$$

Esta es una definición recursiva de la función exponencial 2^n, para definir su valor en $n+1$ se recurre a su valor en n pero con nuestros medios es imposible implementar tal definición pues la función $\lambda n.2^n$ no tiene nombre. Si la nombramos la definición se convierte en:

$$\begin{aligned}potd\,0\ &=\ 1\\ potd\,(n+1)\ &=\ 2*(potd\,n)\ \ n>0\end{aligned}$$

que proporciona la implementación deseada.

De manera que nos gustaría agregar a nuestro lenguaje un mecanismo de declaración de funciones con nombre

$$\mathsf{fun}\ f\,(x)\ \Rightarrow\ e \tag{1}$$

A este respecto la primera pregunta que surge es ¿a qué categoría sintáctica pertenece f?, en la práctica se trata de un identificador y dado que en lenguajes de programación reales no hay

distinción entre los nombres para funciones y los nombres para variables, dado que ambos caen en la categoría de identificadores, resulta adecuado considerar a f como una variable.

Una primera propuesta para nombrar funciones es definir una expresión `let` para declarar funciones con nombre, usando para esto una expresión `let` ordinaria de la siguiente forma:

$$\texttt{let fun } f\,(x) \;\Rightarrow\; e_1 \texttt{ in } e_2 \texttt{ end} =_{def} \texttt{let } f = (\texttt{fun}(x) \;\Rightarrow\; e_1) \texttt{ in } e_2 \texttt{ end}$$

Obsérvese el uso de azúcar sintáctica para definir más azúcar sintáctica, recordemos que `let` es azúcar para $(\lambda x.e_2)\,e_1$. Por ejemplo la función doble puede declarse como sigue

$$\texttt{let fun } \mathit{doble}\,(x) \;\Rightarrow\; x + x \texttt{ in } \mathit{doble}\,2 \texttt{ end}$$

cuya evaluación es:

$$\texttt{let fun } \mathit{doble}\,(x) \;\Rightarrow\; x + x \texttt{ in } \mathit{doble}\,2 \texttt{ end} =_{def} (\lambda\, \mathit{doble}.\mathit{doble}\,2)(\lambda x.x + x)$$

$$(\lambda\, \mathit{doble}.\mathit{doble}\,2)(\lambda x.x + x) \to \big(\mathit{doble}\,2\big)[\mathit{doble} := (\lambda x.x + x)]$$

$$\to (\lambda x.x + x)2 \to 2 + 2 \to 4$$

Aparentemente nuestra propuesta funciona. Sin embargo, además de que nos gustaría tener una forma directa de declaración de funciones con nombre como la dada en (1), en el caso de que la función definida sea recursiva, nuestro mecanismo es defectuoso. Veamos un ejemplo.

$$\texttt{let fun } fac\,(x) \;\Rightarrow\; \textsf{if iszero}\,x \textsf{ then } 1 \textsf{ else } x * fac(x-1) \texttt{ in } fac\,2 \texttt{ end}$$

Este programa pretende definir la función factorial en el argumento 2, la cual debe evaluarse a 2. Veamos el proceso de reducción:

$$\texttt{let fun } fac\,(x) \;\Rightarrow\; \textsf{if iszero}\,x \textsf{ then } 1 \textsf{ else } x * fac(x-1) \texttt{ in } fac\,2 \texttt{ end} =_{def}$$

$$\big(\lambda\, fac.fac\,2\big)\big(\lambda x.\textsf{if iszero}\,x \textsf{ then } 1 \textsf{ else } x * fac(x-1)\big) \to$$

$$(fac\,2)\big[fac := \big(\lambda x.\textsf{if iszero}\,x \textsf{ then } 1 \textsf{ else } x * fac(x-1)\big)\big] =$$

$$\big(\lambda x.\textsf{if iszero}\,x \textsf{ then } 1 \textsf{ else } x * fac(x-1)\big)2 \to$$

$$\textsf{if iszero}\,2 \textsf{ then } 1 \textsf{ else } 2 * fac(2-1) \to$$

$$\textsf{if false then } 1 \textsf{ else } 2 * fac(2-1) \to$$

$$2 * fac(2-1) \not\to$$

Obsérvese que el ligado de fac solo tiene alcance en la expresión $fac\,2$, de tal manera que la siguiente aparición de fac en la segunda expresión es una variable libre, por lo que ya no hay forma de sustituirla por su definición. Seguir evaluando la expresión $2 * fac(2-1)$ nos llevará a un estado bloqueado y jamás e alcanzará el resultado deseado 2.

6. Lisp y las funciones recursivas

Haciendo un poco de historia, Mitchell cuenta[5], que las funciones recursivas eran nuevas cuando Lisp apareció. McCarthy, además de incluirlas en Lisp, fue el principal promotor para agregar funciones recursivas a Algol 60. Fortran, en comparación, no permitía que una función se llamara a si misma. Los miembros del comité para Algol 60 escribieron más tarde que no tenían idea de porqué aceptaron la inclusión de funciones recursivas en Algol 60, tal vez estuvieron molestos algunos años, pero seguramente hoy aceptarían que fue una decisión visionaria.
En su artículo de 1960, *Recursive functions of symbolic expresions and their computation by machine, part I. Communications of the ACM 3(4), pp. 184-195*, McCarthy argumenta que la notación lambda es inadecuada para expresar funciones recursivas. Esta afirmación es falsa. El cálculo lambda puro y por lo tanto Lisp, es capaz de expresar funciones recursivas sin utilizar ningún operador adicional. Este hecho era conocido por los expertos en cálculo lambda en la decada de 1930, pero aparentemente McCarthy y su grupo lo desconocían.

Recordemos que el cálculo lambda puro permite la autoaplicación con la que podemos definir combinadores de punto fijo[6].

Definición 1 *Un λ-término cerrado F es un combinador de punto fijo syss cumple alguna de las siguientes condiciones:*

1. $Fg \to^\star_\beta g(Fg)$
2. $Fg =_\beta g(Fg)$, *es decir, existe un término L tal que* $Fg \to^\star_\beta L$ *y* $g(Fg) \to^\star_\beta L$.

Obsérvese que el caso 1 es un caso particular del 2 pero es de importancia distinguirlo.

Es facil ver que para definir funciones recursivas basta usar un operador de punto fijo. Por ejemplo la función factorial se programa como sigue:

$$\begin{array}{rcl} fac & = & Fg \quad \text{donde} \\ g & = & \lambda f \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * f(n-1) \end{array}$$

El lector debe cerciorarse de que este es un programa correcto para la función factorial y que en particular $fac\,2 \to^\star 2$.

Algunos operadores de punto fijo son:

- (Curry-Rosser) $Y =_{def} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ cuya semántica operacional al aplicarse a una expresión cualquiera g es:

$$Yg \to (\lambda x.g(xx))(\lambda x.g(xx)) \to g\Big((\lambda x.g(xx))(\lambda x.g(xx))\Big) \leftarrow g(Yg)$$

- (Turing) $V =_{def} UU$, donde $U =_{def} \lambda f \lambda x.x(ffx)$
- (Klop) $K =_{def} LL \ldots L$ (26 veces L), donde

$$L = \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$$

[5]En su libro *Concepts in Programming Languages.*

[6]Recordemos que x es punto fijo de una función F si y sólo si $F(x) = x$

7. El lenguaje ISWIM

El cálculo lambda estudiado hasta ahora es demasiado primitivo para ser considerado un lenguaje de programación, al igual que la máquina de Turing es demasiado primitiva para ser equiparada a una computadora real, aún cuando, desde la matemática, ambos formalismos resultan tan poderosos como un lenguaje de programación o una computadora real. En los años 50 y 60 se descubrió la conección entre los lenguajes de programación y varios aspectos del cálculo lambda, esto debido al deseo particular de especificar el significado del lenguaje ALGOL 60 y de formalizar el estudio de los lenguajes de programación empleando sistemas matemáticos conocidos.

Uno de los primeros lenguajes de programación formales, basado en el cálculo lambda, es ISWIM[7], creado por Peter Landin. Si bien este lenguaje, o más bien esta fámilia de lenguajes, nunca fue implementada directamente, sirvío de base para explorar aplicaciones e implementaciones mediante máquinas abstractas o virtuales. El lenguaje real más cercano a ISWIM es SCHEME.

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 a nuestro lenguaje EAB de expresiones aritméticas y booleanas consiste de

$$\mathcal{C} = \{\mathsf{true}, \mathsf{false}, 0, 1, 2 \ldots\} \qquad \mathcal{O} = \{\mathsf{suc}, \mathsf{pred}, \mathsf{iszero}, \mathsf{if}, \mathtt{suma}, \mathtt{prod}\}$$

La semántica operacional $\rightarrow$ 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:

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

Adicionalmente se requiere las reglas burocráticas que modelan la estrategia de evaluación, en nuestro caso de izquierda a derecha. Si $o^{(n)} \in \mathcal{O}$ entonces agregamos las siguientes reglas δ:

$$\frac{e_1 \rightarrow e_1'}{o(e_1, \ldots, e_n) \rightarrow_\delta o(e_1', \ldots, e_n)}$$

$$\frac{e_2 \rightarrow e_2'}{o(v_1, \ldots, e_n) \rightarrow_\delta o(v_1, \ldots, e_n)}$$

$$\frac{e_n \rightarrow e_n'}{o(v_1, \ldots, v_{n-1}, e_n) \rightarrow_\delta o(v_1, \ldots, v_{n-1}, e_n')}$$

8. Ejercicios

1. De acuerdo a la representación de booleanos en el cálculo lambda realice lo siguiente.

   a) Implementa la función disyunción `or`

   b) Implementa la función disyunción excluxiva `xor`

[7] Un acrónimo para la frase *If you See What I Mean*

c) Implemente la función implicación `imp`

d) Implemente la función equivalencia `equiv`

e) Muestre que sus definiciones son correctas en cualesquiera dos valores booleanos, es decir, muestre que para cualesquiera b_1, b_2 booleanos y cualquier función booleana `f` la aplicación `f` $b_1,\ b_2$ se reducen al valor booleano esperado de acuerdo a la tabla de verdad de `f`.

2. Decimos que dos funciones f, g son β-equivalentes syss para cualquier término r, existe un término t tal que $fr \to_\beta^\star t$ y $gr \to_\beta^\star t$. Muestre que los siguientes pares de funciones son β-equivalentes:

   a) I y $Ap\,(Ap\,I)$ donde $I =_{def} \lambda x.x,\ Ap =_{def} \lambda x.\lambda y.xy$

   b) Ap y $\lambda x \lambda y.\langle x, y\rangle\, I$ donde el par es como se definió en clase.

   c) I y $\omega(\omega\ \mathsf{false})$ recordando que $\omega = \lambda x.xx$

3. Implemente la estructura de datos terna de manera análoga a los pares, mediante la definición de las siguientes funciones:

   a) `mktr`, tal que `mktr` $r\,s\,t$ construye la terna $\langle r, s, t\rangle$

   b) `tfst`, la primera proyección tal que `tfst` $\langle r, s, t\rangle \to^\star r$

   c) `tsnd`, la segunda proyección tal que `tsnd` $\langle r, s, t\rangle \to^\star s$

   d) `tthr`, la tercera proyección tal que `tthr` $\langle r, s, t\rangle \to^\star t$

   Debe mostrar que sus definiciones cumplen con las reducciones especificadas.

4. Con respecto a las funciones sucesor y predecesor en numerales de Church realice lo siguiente:

   a) Evalúe $\mathsf{suc}(\mathsf{pred}\,\bar{0})$

   b) Evalúe $\mathsf{suc}(\mathsf{pred}\,\bar{1})$

   c) ¿ A qué se evalúa $\mathsf{suc}(\mathsf{pred}\,\bar{n})$ para un numeral arbitrario $\bar{n}$?

   d) ¿ Qué se puede decir de la evaluación de $\mathsf{suc}(\mathsf{pred}\,r)$ para un término arbitrario r ?

   e) Conteste todos los incisos anteriores para $\mathsf{pred}(\mathsf{suc}\,x)$

5. Implemente las siguientes funciones para numerales de Church:

   a) Relación de orden `lq`

   b) Igualdad de numerales `eq`

   c) Diferencia positiva `dp` (si $n < m$, `dp` n m debe devolver 0)

   d) Cociente y residuo entre dos numerales.

6. Una manera de implementar listas en el cálculo lambda es mediante pares, la lista $x : xs$ se implementa mediante el par $\langle x, xs\rangle$. De esta manera se tiene que

$$[x_1, \ldots, x_k] =_{def} \langle x_1, \langle x_2, \ldots, \langle x_n, Nil\rangle \ldots\rangle\rangle$$

   Con esta idea en mente implemente las siguientes funciones:

   a) Lista vacía: `nil`
   b) Constructor de listas `cons`
   c) Cabeza: `head`
   d) Cola: `tail`
   e) Test de lista vacía: `isnil`

7. Otra manera de implementar listas en el cálculo lambda es mediante la función de iteración fold, la idea es representar una lista $[x_1, \ldots, x_n]$ como:

$$[a_1, \ldots, a_n] =_{def} \lambda f.\lambda x.fa_1(fa_2(\ldots fa_{n-1}(fa_n x)\ldots))$$

   a) Explique la idea de esta implementación en analogía a los numerales de Church.
   b) Implemente las listas $[1], [2, 3], [4, 5, 6]$
   c) Implemente las siguientes funciones:
      1) Lista vacía: `nil`
      2) Constructor de listas `cons`
      3) Cabeza: `head`
      4) Cola: `tail` (esta corresponde al predecesor por lo que resulta más complicada que las demás)
      5) Test de lista vacía: `isnil`

8. Con respecto a la implementación de listas defina recursivamente con ayuda del combinador de punto fijo las siguientes funciones

   a) Longitud: `length`
   b) Concatenación: `app`
   c) Reversa: `rev`

9. Defina recursivamente con ayuda del combinador de punto fijo las siguientes funciones aritméticas:

   a) Suma
   b) Producto
   c) Exponenciación.