Universidad Nacional Autónoma de México

Facultad de Ciencias

Lenguajes de Programación

Tarea 3: Cálculo Lambda

Johann Ramón Gordillo Guzmán
418046090

José Jhovan Gallardo Valdéz
310192815

Tarea presentada como parte del curso de **Lenguajes de Programación** impartido por la profesora **M.I. Karla Ramírez Pulido**.

23 de Marzo del 2020

Link al código fuente: https://github.com/JohannGordillo/

1. Preguntas

1. Currifica cada uno de los siguientes términos:

   a) $\lambda abc.abc$

   Respuesta.

   $\lambda abc.abc \ \rightarrow \lambda a.\lambda b.\lambda c.abc$

   b) $\lambda abc.\lambda cde.acbdce$

   Respuesta.

   $\lambda abc.\lambda cde.acbdce \rightarrow \lambda a.\lambda b.\lambda c.\lambda c.\lambda d.\lambda e.acbdce$

   c) $(\lambda x.\ (\lambda xy.y)\ (\lambda zw.w))\ (\lambda uv.v)$

   Respuesta.

   $(\lambda x.\ (\lambda xy.y)\ (\lambda zw.w))\ (\lambda uv.v) \rightarrow (\lambda x.\ (\lambda x.\lambda y.y)\ (\lambda z.\lambda w.w))\ (\lambda u.\lambda v.v)$

2. Para cada uno de los siguientes términos, aplica α-conversiones para obtener términos donde todas las variables de ligado sean distintas.

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

   Respuesta.

$$
\begin{aligned}
\lambda x.\lambda y.\ (\lambda x.y\ \lambda y.x) &\equiv_\alpha \lambda z.\lambda y.\ (\lambda x.y\ \lambda y.x)[x := z] \\
&\equiv_\alpha \lambda z.\lambda y.\ (\lambda x.y\ \lambda y.z) \\
&\equiv_\alpha \lambda z.\lambda w.\ (\lambda x.y\ \lambda y.z)[y := w] \\
&\equiv_\alpha \lambda z.\lambda w.\ (\lambda x.w\ \lambda y.z)
\end{aligned}
$$

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

   Respuesta.

$$
\begin{aligned}
\lambda x.\ (x\ (\lambda y.\ (\lambda x.x\ y)\ x)) &\equiv_\alpha \lambda z.\ (x\ (\lambda y.\ (\lambda x.x\ y)\ x))[x := z] \\
&\equiv_\alpha \lambda z.\ (z\ (\lambda y.\ (\lambda x.x\ y)\ z))
\end{aligned}
$$

c) $\lambda a.\ (\lambda b.a\ \lambda b.\ (\lambda a.a\ b))$

Respuesta.

$$\begin{aligned}
\lambda a.\ (\lambda b.a\ \lambda b.\ (\lambda a.a\ b)) &\equiv_\alpha \lambda x.\ (\lambda b.a\ \lambda b.\ (\lambda a.a\ b))[a := x] \\
&\equiv_\alpha \lambda x.\ (\lambda b.x\ \lambda b.\ (\lambda a.a\ b)) \\
&\equiv_\alpha \lambda x.\ (\lambda y.x[b := y]\ \lambda b.\ (\lambda a.a\ b)) \\
&\equiv_\alpha \lambda x.\ (\lambda y.x\ \lambda b.\ (\lambda a.a\ b)) \\
&\equiv_\alpha \lambda x.\ (\lambda y.x\ \lambda z.\ (\lambda a.a\ b)[b := z]) \\
&\equiv_\alpha \lambda x.\ (\lambda y.x\ \lambda z.\ (\lambda a.a\ z))
\end{aligned}$$

3. Aplicar las β-reducciones correspondientes a las siguientes expresiones hasta llegar a una Forma Normal o justificar por qué dicha forma no existe. Indicar en cada paso la redex y el reducto. Considerar las siguientes definiciones:

$I =_{def} \lambda x.x$ $\qquad$ $S =_{def} \lambda x.\lambda y.\lambda z.xz\ (yz)$

$K =_{def} \lambda x.\lambda y.x$ $\qquad$ $\Omega =_{def} (\lambda x.xx)\ (\lambda x.xx)$

a) $\lambda x.xK\Omega$

Respuesta.

Para esta expresión no existe una Forma Normal, pues la expresión Ω diverge. Esto es:

$$\begin{aligned}
(\lambda x.xx)\ (\lambda x.xx) &\to_\beta xx[x := \lambda x.xx] \\
&\to_\beta (\lambda x.xx)\ (\lambda x.xx) \\
&\to_\beta \Omega
\end{aligned}$$

b) $(\lambda x.x\ (II))\ z$

Respuesta.

$$\begin{aligned}
(\lambda x.x\ (II))\ z &\to_\beta (\lambda x.x\ (\lambda x.x\ \lambda x.x))\ z \\
&\to_\beta (\lambda x.x\ (x[x := \lambda x.x]))\ z \\
&\to_\beta (\lambda x.x\ (\lambda x.x))\ z \\
&\to_\beta (x[x := \lambda x.x])\ z \\
&\to_\beta (\lambda x.x)\ z \\
&\to_\beta x[x := z] \\
&\to_\beta z
\end{aligned}$$

c) $(\lambda u.\lambda v.\ (\lambda w.w\ (\lambda x.xu))\ v)\ y\ (\lambda z.\lambda y.zy)$

Respuesta.

$$
\begin{aligned}
(\lambda u.\lambda v.\ (\lambda w.w\ (\lambda x.xu))\ v)\ y\ (\lambda z.\lambda y.zy) &\to_\beta (\lambda v.\ (\lambda w.w\ (\lambda x.xu))[u := v])\ y\ (\lambda z.\lambda y.zy)\\
&\to_\beta (\lambda v.\ (\lambda w.w\ (\lambda x.xv)))\ y\ (\lambda z.\lambda y.zy)\\
&\to_\beta (\lambda w.w\ (\lambda x.xv)[v := y])\ (\lambda z.\lambda y.zy)\\
&\to_\beta (\lambda w.w\ (\lambda x.xy))\ (\lambda z.\lambda y.zy)\\
&\to_\beta (w[w := \lambda x.xy])\ (\lambda z.\lambda y.zy)\\
&\to_\beta (\lambda x.xy)\ (\lambda z.\lambda y.zy)\\
&\to_\beta xy[x := \lambda z.\lambda y.zy]\\
&\to_\beta (\lambda z.\lambda y.zy)\ y\\
&\to_\beta \lambda y.zy[z := y]\\
&\to_\beta \lambda y.yy
\end{aligned}
$$

d) $S\ (KI)\ (KI)$

Respuesta.

$$
\begin{aligned}
S\ (KI)\ (KI) &\to_\beta ((\lambda x.\lambda y.\lambda z.xz\ (yz))\ ((\lambda x.\lambda y.x)(\lambda x.x)))\ ((\lambda x.\lambda y.x)(\lambda x.x))\\
&\to_\beta ((\lambda x.\lambda y.\lambda z.xz\ (yz))\ ((\lambda x.\lambda y.x)(\lambda x.x)))\ (\lambda y.x[x := \lambda x.x])\\
&\to_\beta ((\lambda x.\lambda y.\lambda z.xz\ (yz))\ ((\lambda x.\lambda y.x)(\lambda x.x)))\ (\lambda y.\lambda x.x)\\
&\to_\beta ((\lambda x.\lambda y.\lambda z.xz\ (yz))\ (\lambda y.x[x := \lambda x.x]))\ (\lambda y.\lambda x.x)\\
&\to_\beta ((\lambda x.\lambda y.\lambda z.xz\ (yz))\ (\lambda y.\lambda x.x))\ (\lambda y.\lambda x.x)\\
&\to_\beta ((\lambda y.\lambda z.xz[x := yz])\ (\lambda y.\lambda x.x))\ (\lambda y.\lambda x.x)\\
&\to_\beta ((\lambda y.\lambda z.(yz)z)\ (\lambda y.\lambda x.x))\ (\lambda y.\lambda x.x)\\
&\to_\beta (\lambda z.(yz)z[y := \lambda y.\lambda x.x])\ (\lambda y.\lambda x.x)\\
&\to_\beta (\lambda z.((\lambda y.\lambda x.x)z)z)\ (\lambda y.\lambda x.x)\\
&\to_\beta (\lambda z.(\lambda x.x[y := z])z)\ (\lambda y.\lambda x.x)\\
&\to_\beta (\lambda z.(\lambda x.x)z)\ (\lambda y.\lambda x.x)\\
&\to_\beta (\lambda x.x[z := z])\ (\lambda y.\lambda x.x)\\
&\to_\beta (\lambda x.x)\ (\lambda y.\lambda x.x)\\
&\to_\beta x[x := \lambda y.\lambda x.x]\\
&\to_\beta \lambda y.\lambda x.x
\end{aligned}
$$

4. De acuerdo a la representación de números (Numerales de Church) y representación de booleanos en el Cálculo λ:

   a) Define una función $<$ que decide si un número es menor a otro.

      Respuesta.

      Sabemos que el operador relacional $\geq$ se define como:

      $\geq =_{def} \lambda x.\lambda y.z(xPy)$ donde P es la función predecesor.

      Así, dados dos números $x,\ y$ tenemos que x es menor que y si y solamente si x no es mayor o igual que y, por lo que podemos definir al operador relacional $<$ como:

      $< =_{def} \lambda x.\lambda y.(neg\ (\geq x\ y))$

   b) Define la función equivalencia ($\leftrightarrow$) sobre booleanos.

      Respuesta.

      Podemos definir la función consecuencia y con base en ella y la función conjunción podemos definir posteriormente la función equivalencia, ya que $x \leftrightarrow y \equiv (x \rightarrow y) \wedge (y \rightarrow x)$

      Tenemos entonces:

      $\rightarrow =_{def} \lambda x.\lambda y.xyT$

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

      Por lo que la función equivalencia se define como:

      $\leftrightarrow =_{def} \lambda x.\lambda y.(\wedge\ (\rightarrow x\ y)\ (\rightarrow y\ x))$

   c) Define la función disyunción exclusiva xor sobre booleanos.

      Respuesta.

      En pseudocódigo, la función xor de $x,\ y$ se define como:

      ```
      if x:
          not y
      else:
          y
      ```

      Por lo que usando codificación de Church, y sabiendo que la función neg para la negación booleana se define como $neg =_{def} \lambda x.x.FT$ podemos definir a la función xor de x y y como:

$xor =_{def} \lambda x.\lambda y.x(not\ y)y$

5. Dada la siguiente expresión en Racket:

```
(let ([sum (λ (n) (if (zero? n) 0 (+ n (sum (sub1 n)))))])
        (sum 5))
```

   a) Ejecútala y explica el resultado.

   Respuesta.

   Esta función lo que hace es que dado un número n, regresa la suma de los primeros n números naturales. Es decir, regresa $n + (n-1) + ... + 2 + 1 + 0$. O bien, $\frac{n(n+1)}{2}$.

   Por lo que al ejecutar la función recursiva con un parámetro $n = 5$, obténemos:

$$\begin{aligned} (sum\ 5) &= 5 + (sum\ 4) \\ &= 5 + 4 + (sum\ 3) \\ &= 5 + 4 + 3 + (sum\ 2) \\ &= 5 + 4 + 3 + 2 + (sum\ 1) \\ &= 5 + 4 + 3 + 2 + 1 + (sum\ 0) \\ &= 5 + 4 + 3 + 2 + 1 + 0 \\ &= 15 \end{aligned}$$

   $\therefore (sum\ 5) = 15$

   Más formalmente, se puede definir a la función recursiva SUM como sigue:

   $SUM =_{def} \lambda f.\lambda n.if\ n = 0\ then\ 0\ else\ n + (ff)(n-1)$

   Lo que al ejecutarla con un parámetro $n = 5$, da:

$$\begin{aligned} (SUM\ SUM)\ 5 &=_{def} ((\lambda f.\lambda n.if\ n = 0\ then\ 0\ else\ n + (ff)(n-1))\ SUM)\ 5 \\ &\rightarrow_\beta (\lambda n.if\ n = 0\ then\ 0\ else\ n + (SUM\ SUM)(n-1))\ 5 \\ &\rightarrow_\beta if\ 5 = 0\ then\ 0\ else\ 5 + (SUM\ SUM)\ 4 \\ &\rightarrow_\beta 5 + (SUM\ SUM)\ 4 \\ &=_{def} 5 + ((\lambda f.\lambda n.if\ n = 0\ then\ 0\ else\ n + (ff)(n-1))\ SUM)\ 4) \\ &\rightarrow^*_\beta 15 \end{aligned}$$

   b) Modificala usando el Combinador de Punto Fijo Y. Ejecútala y explica el resultado.

   Respuesta.

   Por definición, el combinador de punto fijo Y es:

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

Para usar el Combinador Y, debemos adaptar nuestra definición, de manera que la función reciba una función como parámetro, tal como se hizo en el primer inciso pero implementado en *Racket*:

```
(let ([sum (λ(sum) (λ (n) (if (zero? n) 0 (+ n (sum (sub1 n))))))])
          (sum 5))
```

De esta forma, podemos definir a Y mediante un identificador y aplicarlo a sum. Usando let* para faciliar la escritura, se tiene que:

```
(let* ([Y (λ(f) ((λ(x) (f (x x))) (λ(x) (f (x x)))))]
          [sum (Y (λ(sum) (λ(n) (if (zero? n) 0 (+ n (sum (sub1 n)))))))])
                    (sum 5))
```

Lo que al ejecutarlo en *Racket* se cicla, debido a que el regímen de evaluación del lenguaje es glotón.

c) Modificala usando el Combinador de Punto Fijo Z. Ejecútala y explica el resultado.

Respuesta.

Por definición, el combinador de punto fijo Z es:

$Z =_{def} \lambda g.(\lambda x.g(\lambda v.xxv))(\lambda x.g(\lambda v.xxv))$

Para usar el Combinador Z, debemos adaptar nuestra definición, de manera que la función reciba una función como parámetro, tal como se hizo en el primer inciso pero implementado en *Racket*:

```
(let ([sum (λ(sum) (λ (n) (if (zero? n) 0 (+ n (sum (sub1 n))))))])
          (sum 5))
```

De esta forma, podemos definir a Z mediante un identificador y aplicarlo a sum. Usando let* para faciliar la escritura, se tiene que:

```
(let* ([Z (λ(g) ((λ(x) (g (λ(v) ((x x) v)))) (λ(x) (g (λ(v) ((x x) v))))))]
          [sum (Z (λ(sum) (λ(n) (if (zero? n) 0 (+ n (sum (sub1 n)))))))])
                    (sum 5))
```

Lo que al ejecutarlo da como resultado 15. El uso de este combinador de punto fijo funciona en *Racket* debido a que tiene un regímen de evaluación glotón.

2. Bibliografía

- Ramírez, K. (2020).
  Notas del curso de Lenguajes de Programación.
  Facultad de Ciencias - UNAM
  Ciudad de México, México.

- Krishnamurthi, S. (2017).
  Programming Languages: Application and Interpretation.
  Estados Unidos de América.

- Sookocheff, K. (2018).
  Alpha Conversion.
  Recuperado el 23 de marzo del 2020 de:
  https://sookocheff.com/post/fp/alpha-conversion/

- Sookocheff, K. (2018).
  Beta Reduction.
  Recuperado el 23 de marzo del 2020 de:
  https://sookocheff.com/post/fp/beta-reduction/