Lenguajes de Programación, 2022-1
Nota de clase 07: Combinador Y

Karla Ramírez Pulido Manuel Soto Romero Javier Enríquez Mendoza

29 de noviembre de 2021
Facultad de Ciencias UNAM

En la Nota de Clase 6, revisamos la implementación de un intérprete recursivo mediante el uso de la estructura *caja* y la definición de ambientes recursivos. En esta nota revisaremos una alternativa a esta implementación, basádonos en un formalismo del Cálculo λ llamado *Combinador de punto fijo Y* o simplemente *Combinador Y*[1].

7.1. Cálculo λ y Recursión

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

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

Observación 7.1. *Abusamos de la notación incluyendo primitivas como if o números con fines didácticos.*

Al tratar de evaluar esta definición, la variable f es una variable libre, lo cual representa un problema, pues en tiempo de ejecución no deben de existir variables libres.

$$\begin{aligned} FACT\ 3 &=_{def} (\lambda n.if\ n = 0\ then\ 1\ else\ n * f\ (n - 1))\ 3 \\ &\to_\beta if\ 3 = 0\ then\ 1\ else\ 3 * f\ 2 \\ &\to_\beta 3 * f\ 2 \\ &\to_\beta \text{¿?} \end{aligned}$$

Para ligar la variable f, podemos redefinir la función de forma que f 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.

[1] *Y-Combinator*.

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

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

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

Esta forma de definir funciones recursivas, se puede aplicar para cualquier función, no únicamente para la definición de factorial. Sin embargo es tediosa y si no se aplica correctamente es propensa a errores. Existe una función que realiza esta conversión de manera automática llamada *Combinador de punto fijo Y*.

A continuación se definen los conceptos necesarios para definir y entender esta función.

7.2. Combinadores

Un combinador es una expresión cerrada, es decir, que no tiene variables libres. Permiten la definición de nuevas funciones. Existen una infinidad de combinadores dentro del Cálculo λ, por ejemplo [1]:

- La identidad es el combinador más simple y uno de los más utilizados. Se representa por I y se define de la siguiente manera:

$$\lambda x.x$$

- La función constante que sin importar el parámetro que reciba siempre regresa una variable x. Se utiliza para definir funciones constantes y se representa por K:

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

- Una versión generalizada de la aplicación de función es el combinador S:

$$\lambda x.\lambda y.\lambda z.\,(xz\,(yz))$$

Algo interesante de estos dos últimos combinadores, es que existe una completud con los combinadores K y S. Esto quiere decir que usando estos dos combinadores es posible producir cualquier otro combinador del Cálculo λ. Por ejemplo, podremos definir el combinador I a partir de S y K.

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

7.3. El Combinador Y

En las matemáticas, se dice que x es *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* [2].

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:

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

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

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

Definición 7.1. (Combinador de punto fijo Y)

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

Teorema 7.1. *Y es un combinador de punto fijo.*

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

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

□

Se deja como ejercicio al lector, probar que en efecto $(Y\ FACT)\ 3$ da como resultado 6.

7.4. Implementando Recursión con Y

Usando la definición del Combinador Y, es posible modificar el intérprete de **`RCFWAE`** de manera tal que permita la recursión de expresiones sin el uso de la primitiva **`rec`**.

Veamos primero cómo funciona en un lenguaje como Racket. Por ejemplo, la siguiente expresión define el factorial de un número usando la primitiva **`let`**.

```
(let ([fact (λ(n) (if (zero? n) 1 (* n (fact (sub1 n)))))])
   (fact 3))
```

Para usar el Combinador Y, debemos adaptar nuestra definición, de manera que la función reciba una función como parámetro.

```
(let ([fact (λ(fact) (λ(n) (if (zero? n) 1 (* n (fact (sub1 n))))))])
   (fact 3))
```

De esta forma, podemos definir `Y` mediante un identificador y aplicarlo a `fact`. Usamos `let*` para faciliar la escritura:

```
(let* ([Y (λ(f) ((λ(x) (f (x x))) (λ(x) (f (x x)))))]
       [fact (Y (λ(fact) (λ(n) (if (zero? n) 1 (* n (fact (sub1 n)))))))])
   (fact 3))
```

Sin embargo, al cargar y ejecutar esta última expresión en DrRacket, se puede apreciar cómo el programa *se cicla* y nunca da un resultado. Esto ocurre principalmente por la definición del Combinador Y y en particular por el régimen de evaluación de Racket.

Analicemos la definición de `Y` en Racket aplicada a una función `g`:

```
> ((λ(f) ((λ(x) (f (x x))) (λ(x) (f (x x))))) g)
> ((λ(x) (g (x x))) (λ(x) (g (x x))))
> (g ((λ(x) (g (x x))) (λ(x) (g (x x)))))
> (g (g ((λ(x) (g (x x))) (λ(x) (g (x x))))))
> (g (g (g ((λ(x) (g (x x))) (λ(x) (g (x x)))))))
...
```

Como se puede apreciar cada llamada aplica el combinador de punto fijo, de forma que en cada paso se genera una nueva aplicación de `g`, generando infinitas llamadas. Esto ocurre pues llamada tras llamada el parámetro real de la función se evalúa aunque no sea necesario. Esta evaluación ocurre así pues Racket usa un régimen de evaluación glotón.

Este problema fue descubierto durante el diseño de ISWIM pues el Cálculo λ usa un régimen de evaluación perezoso mediante una técnica de paso de parámetros *por nombre*, mientras que `ISWIM` usaba un régimen de evaluación glotón mediante una técnica de paso de parámetros *por valor*.

Como mencionamos en la Nota de Clase 3, ISWIM nunca llegó a ser terminado, sin embargo Scheme es el lenguaje más cercano y parecido a éste, por lo que Racket hereda también este comportamiento.

Para corregir este detalle del Combinador Y, existe otro combinador de punto fijo, llamado Z[2], que usa la técnica de paso de parámetros por valor y se define como sigue:

Definición 7.2. (Combinador de punto fijo Z)

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

[2]Algunos autores lo llaman Y_v, la v viene del paso de parámetros por valor.

Modificando nuestra expresión para que use Z, tenemos:

```
(let* ([Z (λ(g) ((λ(x) (g (λ(v) ((x x) v)))) (λ(x) (g (λ(v) ((x x) v))))))]
       [fact (Z (λ(fact) (λ(n) (if (zero? n) 1 (* n (fact (sub1 n)))))))])
   (fact 3))
```

Que en al ejecutar en DrRacket da como resultado 6.

7.4.1. Implementación

Podemos usar esta definición para **RCFWAE** mediante los pasos siguientes:

1. Considerar la primitiva **rec** como azúcar sintáctica de **with**, añadiendo la aplicación de **Z** al principio del valor asociado al identificador correspondiente y añadiendo un nuevo parámetro (currificado) a la función que permita la autoreferencia. Esto es

   ```
   {rec {<id> <value>}
      <body>}
   ```

   se transforma en

   ```
   {with {<id> {Z {fun {<id>} <value>}}}
      <body>}
   ```

2. Al llamar al intérprete, añadir al ambiente de evaluación inicial la definición de **Z**.

   ```
   (interp <expr> (aSub 'Z ...))
   ```

Observación 7.2. El uso de Z para implementar un comportamiento recursivo, sólo debe usarse para lenguajes con un régimen de evaluación glotón (al igual que en ISWIM). En el caso de lenguajes que hacen uso de la evaluación perezosa, puede usarse sin problemas la definición de Y (al igual que en el Cálculo-λ).

7.5. Ejercicios

Dada la siguiente expresión, modifícala para que use recursión correctamente usando usando el Combinador Y. ¿Es posible hacerlo? ¿Por qué? ¿Cómo se corrige?:

```
(let ([suma (λ(l)
              (if (empty? l) 0 (+ (car l) (suma (cdr l)))))])
   (suma '(1 2 3)))
```

Referencias

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

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

[3] Éric Tanter, *A Note on Recursion*, 2016.

[4] Richard P. Gabriel, *The Why of Y*, Lucid, Inc. and Stanford University. 2001.