Combinador Y

Lenguajes de programación

Alumnos:
Lenin Quetzal Vázquez Merino
Emiliano Kaleb Jiménez Rivera

Materia: Lenguajes De Programación

Profesor: Dr. Manuel Soto Romero

1 de diciembre del 2025

Índice

1. Prefacio **4**
1.1. Agradecimientos 4

2. Introducción **4**

3. Teoría del punto fijo **4**
3.1. Teorema del punto fijo de Brouwer 5
3.2. Teorema del punto fijo de Schauder 5

4. Combinador Y **5**

5. Autointepretación **5**

6. Autointepretación usando un combinador Y **6**

7. Conclusión **6**

8. Bibliografías **7**

1. Prefacio

Este documento está dedicado a aquellas personas que tengan un interes por la programación, sin embargo, se concidera que el lector tiene conocimiento basico o esta familiarizado con el calculo lambda.

1.1. Agradecimientos

Se agradece epecialmente a los tutores del curso: Dr. Manuel Soto Romero, Diego Méndez Medina y Erick Daniel Arroyo Martínez, por el apoyo brindado a lo largo del curso. También se le agradece los ayudantes y compañeros por leer el documento.

2. Introducción

El cálculo lambda fue inventado por el matemático estadounidense Alonzo Church alrededor del año 1930; él lo introdujo como un sistema formal para describir la computación. Este modelo se caracterizó por sentar las bases para los lenguajes funcionales.

Para 1958, **Lisp** inició como un proyecto liderado por John McCarthy, convirtiéndolo en el primer lenguaje funcional de la historia. Algo que caracterizó a Lisp fue la capacidad de definir funciones recursivas. Sin embargo, McCarthy encontró un problema con las funciones, el cual consistía en que las variables libres de una función se determinaban buscando en el ambiente de llamada y no en la función donde se definían, lo que causaba confusión ya que no era nada intuitivo.
Por ejemplo, dada la siguiente expresión:

```
 ( let (x 3)
( let (f ( lambda (y) (+ x y )))
( let (x 5)
(f 4))))
```

1. Al principio se define $x = 3$.
2. Luego se define la función f.
3. Posterior se define $x = 5$

Entonces, al evaluar (f 4), se sustituye y por 4 para así obtener $(+x\,4)$. Pero, al buscar x, se halla un 5 (resultado de buscar desde el ambiente de llamada), por lo que $(\text{f } 4) = (+5\,4) = 9$. Si se observa con cuidado, se podrá notar que el valor intuitivo a obtener es $(\text{f } 4) = (+3\,4) = 7$, debido a que la variable x de la función f está dentro del alcance de la variable $x = 3$.

De esto surgió un concepto nuevo, conocido como **cerradura** (o clausura), que logró resolver este problema a la perfección. Pero así de bien como lo hizo, también originó uno nuevo: la recursión que caracterizaba a este programa se perdía al usar cerraduras. Entonces surgió la pregunta: ¿Cómo lograr la recursión?

3. Teoría del punto fijo

Un punto x es un punto **fijo** en el dominio de una función f si $f(x) = x$. Por ejemplo: si $f(x) = x^2$, entonces $x = 1$ y $x = 0$ son puntos fijos, ya que $f(1) = 1$ y $f(0) = 0$.

Dos de los teoremas más importantes de esta teoría son:

3.1. Teorema del punto fijo de Brouwer

El teorema establece que si $f : B \to B$ es una función continua y B es una bola en R^n, entonces f tiene un punto fijo.

3.2. Teorema del punto fijo de Schauder

Si B es un subconjunto compacto y convexo de un espacio de Banach X y $f : B \to B$ es una función continua, entonces f tiene un punto fijo.

Ahora bien, la teoría del punto fijo es importante porque la idea marca el inicio de la recursión en los lenguajes. En particular, en el cálculo lambda es donde logra mayor expresividad.

4. Combinador Y

Alrededor del año **1930**, **Haskell Brooks Curry**, matemático estadounidense, se enfocaba en la **lógica combinatoria** y buscaba un medio para construir funciones recursivas.
Utilizando la teoría del **punto fijo** dentro del marco del **cálculo lambda sin tipos** o la lógica combinatoria, él necesitaba encontrar un **combinador Y** tal que, para cualquier función g, el resultado $\mathbf{Y}g$ fuera un punto fijo de g. Es decir:

$$\mathbf{Y}g = g(\mathbf{Y}g)$$

La expresión que se le atribuye a **Curry** es una de las formulaciones más conocidas para el **combinador de punto fijo Y**:

$$\mathbf{Y} = \lambda g.(\lambda x.g(x\,x))(\lambda x.g(x\,x))$$

Aunque Curry fue fundamental en la lógica combinatoria y popularizó la idea, esta formulación exacta del combinador **Y** es frecuentemente atribuida al trabajo de **Alonzo Church** (que lo demostró en el cálculo lambda puro), siendo a veces conocida como el **Combinador de Punto Fijo de Church**.

5. Autointepretación

La **metaprogramación** se define como la capacidad de que un programa pueda ser representado por datos, y que los datos puedan ser representados por el programa. Esto está estrechamente relacionado con la autointerpretación, que busca que un programa pueda interpretarse a sí mismo. Un muy buen ejemplo es Lisp, que gracias a su expresividad, permite metaprogramación. Dada esta cualidad de Lisp, da la impresión de que McCarthy vio esto como la eliminación de cualquier limitación derivada del hecho de que el metalenguaje en sí mismo es de primer orden [1]. Sin embargo, como se mencionó en la introducción, se tenía el problema de las variables libres debido a que las funciones eran de primer orden (es decir, que se pueden pasar funciones como argumentos a otras funciones).

A mediados de la década de 1970 Gordon había estado trabajando en un sistema de generación de pruebas llamado LCF para razonar sobre funciones recursivas, en particular en el contexto de los lenguajes de programación. El sistema consistía en un cálculo deductivo llamado PPX

(cálculo de predicados polimórficos) junto con un lenguaje de programación interactivo llamado **ML** metalenguaje (ya que servía como lenguaje de comandos para LCF)[2].
ML se caracterizó por seguir un estilo de programación funcional (aunque ciertamente cuenta con características del estilo imperativo), tomando como modelo al cálculo lambda, pero agregando tipos. Además fue el primer lenguaje en utilizar inferencia de tipos como parte de su semántica e integrar un buen polimorfismo (1988). Un detalle que surgió del lenguaje fue justo en la recursión: ¿cómo habría que tipificar una autoaplicación?, pues el cálculo lambda con tipos restringía el modelo, y hacía imposible utilizar al combinador Y.

A continuación, se muestran las reglas de reducción que implementaba ML.
(1) Typed-α-conversion: $(\lambda x^T_{l'}.e^T) \to (\lambda x^T_{l''}.[x^T_{l'}/x^T_{l''}]e^T)$, donde $x^T_{l''} \notin \text{fv}(e^T)$.
(2) Typed-β-conversion: $(\lambda x^{T_2}.e^{T_1})e^{T_2} \to [e^{T_2}/x^{T_2}]e^{T_1}$
(3) Typed-η-conversion: $\lambda x^{T_1}.(e^{T_2}x^{T_1}) \to e^{T_2}$
si $x^{T_1} \notin \text{fv}(e^{T_2})$. [2]

La solución consistió en ageragar varios operadores de punto fijo constantes(similares a Y), pero tipados.

Typed-Y-conversion: $(\text{Y}^{T\to T}e^{T\to T}) \to e^{T\to T}(\mathbf{Y}^{T\to T}e^{T\to T})$ [2]

Si bien no se hace uso del combinador Y, este sirvió como base para poder lograr la recursión.

Haskell es uno de los lenguajes de programacion funcionales mas recientes, y que surgio alrededor del año 2000, el cuál se caracterizá por:

1. Funciones de primer orden
2. Evaluación perezosa
3. Polimorfismo en funciónes.
4. Ecuaciones y coincidencia de patrones.
5. Semántica formal

(Algunas de estas caracterizticas definen a los programacion funcionales mas actuales)

Este lenguaje usa como base el calculo labda, y dadas sus cualidas, es perfecto para realizar metaprogramación. Por ello en la siguinte sección se ilustraran las ideas primcipales para realizar un pequeño programa que se interprete a si mismo usando el combinador Y.

6. Autointepretación usando un combinador Y

7. Conclusión

Actualmente los lenguajes de programación funcionales modernos se caracterizan por implementar:

1. Funciones de primer orden
2. Evaluación perezosa

3. Abstracción de datos.
4. Ecuaciones y coincidencia de patrones.
5. Semántica formal

Estas características son esenciales y definen al lenguaje. Ejemplo de ello es Haskell, que es muy utilizado por la comunidad ya además de contar con muy buenas características, es bueno en la metaprogramación, ya que permite construir o definir otros lenguajes a partir de sí mismo. Sin embargo no lo es tanto como Lisp, que como vimos resulta capaz de interpretarse a sí mismo.

El combinador Y es una función excelente que nos ayuda a lograr la recursión, pero solo cuando se trata de cálculo lambda puro (de otra forma, es muy difícil que llegue a ser útil), pues en la sección anterior vimos algunos ejemplos donde no funciona. Aunque claro, eso último depende mucho del régimen de evaluación que se tome. Por ejemplo: si el lenguaje es ansioso, entonces es imposible usarlo. Por el contrario, si es perezoso, entonces es ideal.
Entonces regresando a la pregunta de la introducción, la respuesta es que para lograr la recursión es necesario usar un nuevo combinador, llamado combinador z, que se define como:

$$\mathbf{Z} = \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy)) \tag{1}$$

Esto se debe a que el régimen de evaluación que toma Lisp es ansioso.
Por lo tanto, el combinador Y es de utilidad para lograr la autinterpretación en los lenguajes que implementen un régimen perezoso, sin embargo, esto puede llegar a variar dadas las características del lenguaje.

8. Bibliografías

Referencias

[1] D. A. Turner, "Some History of Functional Programming Languages," in *Trends in Functional Programming*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, H.-W. Loidl, and R. Peña, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7829, pp. 1–20, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-40447-4$_1$

[2] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359–411, Sep. 1989. [Online]. Available: https://dl.acm.org/doi/10.1145/72551.72554