Programación funcional

Pedro O. Pérez M., PhD.

Implementación de métodos computacionales
Tecnológico de Monterrey

pperezm@tec.mx

02-2023

Contenido I

- El cálculo lambda fue inventado por Alonzo Church en la década de 1930 para estudiar la interacción de la abstracción funcional y la aplicación de funciones desde un punto de vista abstracto, puramente matemático.
- Fue uno de los muchos sistemas relacionados que se propusieron a finales de los años veinte y treinta. Este fue un momento emocionante para la informática teórica, aunque las computadoras aún no se habían inventado.
- Hubo muchos grandes matemáticos tratando de comprender la noción de computación:
  - Máquinas de Turing - Turing.
  - Funciones m-recursivas - Gödel.
  - Sistemas de reescritura - Post.
  - Cálculo lambda - Church
  - Lógica combinatoria - Schönfinkel, Curry

Todos estos sistemas se veían muy diferentes, pero lo sorprendente fue que todos resultaron ser computacionalmente equivalentes en el sentido de que cada uno podía codificar y simular a los demás. Church pensó que esto era demasiado llamativo para ser una mera coincidencia, y declaró que habían encontrado la definición correcta de la noción de computabilidad: era el espíritu común encarnado de manera diferente por cada uno de estos formalismos. Esta declaración se conoció como la tesis de Church.

El Cálculo Lambda es simple:

- Todo es una función. No hay otros tipos primitivos, no enteros, cadenas, objetos, booleanos, etc. Si quieres estas cosas, debes codificarlas usando funciones.
- Sin efectos secundarios, ni estatales. Es puramente funcional. Por tanto, podemos pensar exclusivamente en términos del modelo de sustitución.
- El orden de evaluación es irrelevante.
- Solo funciones unarias (un argumento).

Los términos lambda $E, F, G, ...$, se definen inductivamente:

- Variables: Cualquier variable $x, y, z, ...$ es un término lambda.
- Aplicación: si E y F son términos lambda, entonces EF es un término lambda.
- Abstracción: Si E es un término lambda y x es una variable, entonces $\lambda.x.E$ es un término lambda.

El cálculo en el Cálculo Lambda se realiza por sustitución. Hay dos reglas principales:

- α-reduction (cambiando el nombre de las variables ligadas): $\lambda x.E \Rightarrow \lambda y.E[x/y]$, donde $E[x/y]$ denota el resultado de reemplazar todas las apariciones libres de x en E por y, siempre que y no sea capturado por un λy ya en E.
- β-reduction (regla de sustitución): $(\lambda x.EF) \Rightarrow E[x/y]$, denota el resultado de reemplazar todas las apariciones libres de x en E por F.

Todas las funciones son unarias en el cálculo lambda. Las funciones de mayor aridad se simulan mediante "currying". Una función de dos argumentos se simula mediante una función de un argumento que devuelve una función de un argumento que devuelve el resultado. Es decir, simularíamos:

```
(lambda (x y) E)
```

en su forma "cirrued" como

```
(lambda (x) (lambda (y) E))
```

Lo que equivale, en notación lambda, a: $\lambda x.\lambda y.E$.

¿Quieres conocer más sobre el cálculo lambda? Revisa la siguiente fuente:

- ROJAS, Raúl. A tutorial introduction to the lambda calculus. arXiv preprint arXiv:1503.09060, 2015.
- "Introducing Lambda Calculus"
- "Lambda Calculus - Computerphile"
- "Curried Functions - Computerphile"

Características generales de la programación funcional

- En equipos de tres, revisen las siguientes páginas Web:
  - "Functional Programming: Concepts, Advantages, Disadvantages, and Applications"
  - "What is functional programming? Explained in Python, JS, and Java"
  - "What is Functional Programming? Tutorial with Example"
- En equipo, responde las siguientes preguntas:
  - ¿Qué es un lenguaje funcional?
  - ¿Cuáles son las características de un lenguaje funcional?
  - ¿Cuáles son sus ventajas?
  - ¿Cuáles son sus desventajas?
  - ¿Qué lenguajes "comerciales" son considerados funcionales?

- La programación funcional es un paradigma de programación en que los programas se construyen aplicando y combinando funciones.
- Trata todas las funciones como funciones matemáticas deterministas o funciones puras, es decir, que siempre devuelven el mismo resultado y no pueden verse afectadas por ningún estado mutable u otros efectos secundarios.
- Los defensores de la programación puramente funcional afirman que al restringir los efectos secundarios, los programas pueden tener menos errores, ser más fáciles de depurar y probar y ser más adecuados para la verificación formal.

Característica 1: Existen funciones de primera clase y de orden superior.

- Las funciones de orden superior son funciones que pueden tomar otras funciones como argumentos o devolverlas como resultados. En cálculo, un ejemplo de una función de orden superior es el operador diferencial d/dx, que devuelve la derivada de una función f.
- Cuando hablamos de funciones de primera clase, nos referimos a entidades del lenguaje de programación que no tienen restricciones en su uso. Por lo tanto, las funciones de primera clase pueden aparecer como argumentos para otras funciones y como sus valores de retorno.
- Las funciones de orden superior permiten la aplicación parcial o el procesamiento, una técnica que aplica una función a sus argumentos de uno en uno, y cada aplicación devuelve una nueva función que acepta el siguiente argumento.

Característica 2: Las funciones son puras.

- Llamamos funciones puras a aquellas funciones que no tienen efectos secundarios (memoria o E/S). Esto significa que las funciones puras tienen varias propiedades útiles, muchas de las cuales pueden usarse para optimizar el código:
  - Si no se utiliza el resultado de una expresión pura, se puede eliminar sin afectar a otras expresiones.
  - Si se llama a una función pura con argumentos que no causan efectos secundarios, el resultado es constante con respecto a esa lista de argumentos, es decir, llamar a la función pura, nuevamente, con los mismos argumentos devuelve el mismo resultado.
  - Si no hay dependencia de datos entre dos expresiones puras, su orden se puede revertir, o se pueden realizar en paralelo y no pueden interferir entre sí.
  - Si todo el lenguaje no permite efectos secundarios, entonces se puede utilizar cualquier estrategia de evaluación; esto le da al compilador la libertad de reordenar o combinar la evaluación de expresiones en un programa.

Característica 3: ¡Son recursivos!

- La iteración en los lenguajes funcionales generalmente se logra mediante recursividad. Las funciones recursivas se invocan a sí mismas, permitiendo que una operación se repita hasta que llegue al caso base.
- En general, la recursividad requiere mantener una pila, que consume espacio en una cantidad lineal hasta la profundidad de la recursividad. Esto podría hacer que la recursividad sea prohibitivamente costosa de usar en lugar de bucles imperativos. Sin embargo, un compilador puede reconocer y optimizar una forma especial de recursividad conocida como recursividad de cola en el mismo código utilizado para implementar la iteración en lenguajes imperativos.
- Los patrones comunes de recursividad se pueden abstraer utilizando funciones de orden superior.

Característica 4: Evaluación estricta vs. no estricta.

- Los lenguajes funcionales pueden clasificarse en función de si utilizan una evaluación estricta (eager) o no estricta (lasy), conceptos que se refieren a cómo se procesan los argumentos de función cuando se evalúa una expresión. La diferencia técnica está en la semántica denotacional de expresiones que contienen cálculos erróneos o divergentes.
- Bajo una evaluación estricta, la evaluación de cualquier término que contenga un subelemento fallido falla.
- La evaluación no estricta no evalúa los argumentos de la función a menos que sus valores sean necesarios para evaluar la propia llamada a la función.
- La estrategia de implementación habitual para la evaluación no estricta en lenguajes funcionales es la reducción de grafos. La evaluación no estricta se utiliza de forma predeterminada en varios lenguajes funcionales puros, incluidos Miranda, Clean y Haskell.

Característica 5: Transparencia referencial.

- Los programas funcionales no tienen declaraciones de asignación, es decir, el valor de una variable en un programa funcional nunca cambia una vez definido. Esto elimina cualquier posibilidad de efectos secundarios porque cualquier variable puede reemplazarse con su valor real en cualquier punto de ejecución. Por tanto, los programas funcionales son referencialmente transparentes.

Ejemplos de lenguajes de programación funcionales:

- LISP.
- **Scheme (https://racket-lang.org/).**
- Haskell.
- Erlang.
- Clojure.