Cálculo lambda

Mario Román

November 9, 2017

Outline

Expresiones λ

Programando en el cálculo λ

Codificación de Church

Expresiones λ

Origen

Sistema lógico inventado por Alonzo Church sobre el 1930, tratando ecuaciones entre expresiones λ. Como lenguaje de programación, considerando expresiones sin ecuaciones, es Turing-completo.

Expresiones lambda

Las expresiones λ pueden ser

$$\texttt{Expr} := \begin{cases} v, & \text{variables, de un conjunto numerable,} \\ \texttt{Expr Expr}, & \text{aplicación de funciones,} \\ \lambda v.\texttt{Expr}, & \text{abstracción sobre una variable.} \end{cases}$$

Ejemplos

- x, la variable x;
- $f\ x$, un término f aplicado a x;
- $(\lambda z.z + z)$, la función que toma un argumento y lo duplica;
- $(\lambda x.x^2 + 1)\ 2$, la función que toma x como argumento y devuelve $x^2 + 1$, aplicada a 2;

β-reducciones

En $(\lambda x.x + x)\ 2$, sabemos que el resultado final debería ser 4.

¿Cómo se ejecuta o se evalúa una expresión λ?

β-reducciones

En $(\lambda x.x + x)\ 2$, sabemos que el resultado final debería ser 4.
¿Cómo se ejecuta o se evalúa una expresión λ?

Beta-reducción

La β-reducción es el proceso de evaluación de una función λ aplicada sobre un argumento. Se realiza una reducción como

$$(\lambda x.M)\ N \quad \longrightarrow_{\beta} \quad M_{[N/x]}$$

β-reducciones

En $(\lambda x.x + x)\ 2$, sabemos que el resultado final debería ser 4.

¿Cómo se ejecuta o se evalúa una expresión λ?

Beta-reducción

La β-reducción es el proceso de evaluación de una función λ aplicada sobre un argumento. Se realiza una reducción como

$$(\lambda x.M)\ N \quad \longrightarrow_\beta \quad M_{[N/x]}$$

donde

- x es la variable sobre la que se abstrae; y M es una expresión dependiente de x, es decir, x puede aparecer dentro de M;

β-reducciones

En $(\lambda x.x + x)\ 2$, sabemos que el resultado final debería ser 4.
¿Cómo se ejecuta o se evalúa una expresión λ?

Beta-reducción

La β-reducción es el proceso de evaluación de una función λ aplicada sobre un argumento. Se realiza una reducción como

$$(\lambda x.M)\ N \quad \longrightarrow_\beta \quad M_{[N/x]}$$

donde

- x es la variable sobre la que se abstrae; y M es una expresión dependiente de x, es decir, x puede aparecer dentro de M;
- $(\lambda x.M)$ es una función λ que toma un argumento y lo sustituye dentro de M;

β-reducciones

En $(\lambda x.x + x)\ 2$, sabemos que el resultado final debería ser 4.
¿Cómo se ejecuta o se evalúa una expresión λ?

Beta-reducción

La β-reducción es el proceso de evaluación de una función λ aplicada sobre un argumento. Se realiza una reducción como

$$(\lambda x.M)\ N \quad \longrightarrow_\beta \quad M_{[N/x]}$$

donde

- x es la variable sobre la que se abstrae; y M es una expresión dependiente de x, es decir, x puede aparecer dentro de M;
- $(\lambda x.M)$ es una función λ que toma un argumento y lo sustituye dentro de M;
- $M_{[N/x]}$ es M, pero con cada x sustituida por N.

Ejemplos de β-reducción

En una reducción, una *abstracción* λ se aplica a un *argumento*

- por ejemplo $(\lambda x.x^2 + 1)\ 3 \quad \longrightarrow_\beta \quad 3^2 + 1$;

Ejemplos de β-reducción

En una reducción, una *abstracción* λ se aplica a un *argumento*

- por ejemplo $(\lambda x.x^2 + 1)\ 3 \quad \longrightarrow_\beta \quad 3^2 + 1$;
- el argumento puede ser una función,
  $(\lambda x.x)(\lambda x.x) \quad \longrightarrow_\beta \quad (\lambda x.x)$

Ejemplos de β-reducción

En una reducción, una *abstracción* λ se aplica a un *argumento*

- por ejemplo $(\lambda x.x^2+1)\ 3 \quad \longrightarrow_\beta \quad 3^2+1$;
- el argumento puede ser una función,
  $(\lambda x.x)(\lambda x.x) \quad \longrightarrow_\beta \quad (\lambda x.x)$
- y una función puede devolver otra
  $(\lambda x.\lambda y.y)\ 3 \quad \longrightarrow_\beta \quad (\lambda y.y)$

Ejemplos de β-reducción

En una reducción, una *abstracción* λ se aplica a un *argumento*

- por ejemplo $(\lambda x.x^2 + 1)\ 3 \quad \longrightarrow_\beta \quad 3^2 + 1$;
- el argumento puede ser una función,
  $(\lambda x.x)(\lambda x.x) \quad \longrightarrow_\beta \quad (\lambda x.x)$
- y una función puede devolver otra
  $(\lambda x.\lambda y.y)\ 3 \quad \longrightarrow_\beta \quad (\lambda y.y)$

Los nombres de las variables son irrelevantes

$$(\lambda x.x^2 + 1) \equiv (\lambda y.y^2 + 1).$$

Primeras definiciones

A partir de ahora, queremos que todo lo que computamos sea mediante expresiones λ. Si por ejemplo queremos trabajar con números, debemos representarlos como expresiones λ.

A partir de ahora, queremos que todo lo que computamos sea mediante expresiones λ. Si por ejemplo queremos trabajar con números, debemos representarlos como expresiones λ.

Primeras definiciones

- $\texttt{id} := (\lambda x.x)$, la función identidad;
- $\texttt{const} := (\lambda x.\lambda y.x)$, la función constante;
- $S := (\lambda x.\lambda y.\lambda z.x\ y\ (x\ z))$, otra función de ejemplo.

Aplicación parcial

Escribimos las funciones y sus argumentos separados por espacios

`const 3 4`

Esto puede interpretarse de dos formas equivalentes:

Aplicación parcial

Escribimos las funciones y sus argumentos separados por espacios

`const 3 4`

Esto puede interpretarse de dos formas equivalentes:

- `const 3 4`, donde `const` es una función en dos argumentos,

Aplicación parcial

Escribimos las funciones y sus argumentos separados por espacios

`const 3 4`

Esto puede interpretarse de dos formas equivalentes:

- `const 3 4`, donde `const` es una función en dos argumentos,
- `(const 3) 4`, donde `const` es una función en un argumento que a su vez devuelve `(const 3)`, otra función en un argumento que se aplica sobre 4.

Aplicación parcial

Escribimos las funciones y sus argumentos separados por espacios

`const` 3 4

Esto puede interpretarse de dos formas equivalentes:

- `const` 3 4, donde `const` es una función en dos argumentos,
- (`const` 3) 4, donde `const` es una función en un argumento que a su vez devuelve (`const` 3), otra función en un argumento que se aplica sobre 4.

Podemos definir funciones parcialmente aplicadas como `siempretres` := `const` 3.

Programando en el cálculo λ

Forma normal

Una expresión está en forma normal si no se le pueden aplicar más β-reducciones. La forma normal es *única* por el teorema de Church-Rosser.

Forma normal

Una expresión está en forma normal si no se le pueden aplicar más β-reducciones. La forma normal es *única* por el teorema de Church-Rosser.

Church-Rosser

Si a un término se le aplican distintas reducciones, los términos que se obtienen pueden reducirse a uno común

$$
\begin{tikzcd}
 & A \arrow[dl, two heads, "\beta"'] \arrow[dr, two heads, "\beta"] & \\
B \arrow[dr, dashed, two heads] & & C \arrow[dl, dashed, two heads] \\
 & D &
\end{tikzcd}
$$

Divergencia

Hemos demostrado que la forma normal es única, pero no que exista o que sepamos encontrarla de alguna forma.

Divergencia

Hemos demostrado que la forma normal es única, pero no que exista o que sepamos encontrarla de alguna forma.

Puede ocurrir que un término no esté en forma normal y sin embargo las reducciones no lo lleven a ella. Por ejemplo,

$$\Omega = (\lambda x.x\ x)(\lambda x.x\ x) \longrightarrow_\beta (\lambda x.x\ x)(\lambda x.x\ x)$$

no llega a forma normal.

Divergencia

Hemos demostrado que la forma normal es única, pero no que exista o que sepamos encontrarla de alguna forma.

Puede ocurrir que un término no esté en forma normal y sin embargo las reducciones no lo lleven a ella. Por ejemplo,

$$\Omega = (\lambda x.x\ x)(\lambda x.x\ x) \longrightarrow_\beta (\lambda x.x\ x)(\lambda x.x\ x)$$

no llega a forma normal. O por ejemplo,

$$(\lambda x.x\ x\ x)(\lambda x.x\ x\ x)$$

se hace más complejo al aplicarle reducciones y diverge.

Evaluación a izquierda

Hay algunas expresiones que llegarán a una forma normal o no dependiendo de cómo los evaluemos

- $\texttt{const id}\ \Omega \quad \longrightarrow_\beta \quad$ `id`, si evaluamos primero `const`;
- $\texttt{const id}\ \Omega \quad \longrightarrow_\beta \quad \texttt{const id}\ \Omega$, evaluando primero Ω.

Podemos demostrar el siguiente teorema.

Evaluación a izquierda

Hay algunas expresiones que llegarán a una forma normal o no dependiendo de cómo los evaluemos

- $\texttt{const id}\,\Omega \quad \longrightarrow_\beta \quad$ `id`, si evaluamos primero `const`;
- $\texttt{const id}\,\Omega \quad \longrightarrow_\beta \quad \texttt{const id}\,\Omega$, evaluando primero Ω.

Podemos demostrar el siguiente teorema.

Evaluación a izquierda

Si existe una forma normal, la estrategia que reduce a cada paso lo más a la izquierda posible la encuentra.

Codificación de Church

Estructuras de datos

Nos gustaría poder describir y usar estructuras de datos dentro del cálculo λ. Cada estructura de datos está definida por un conjunto finito de constructores.

Estructuras de datos

Nos gustaría poder describir y usar estructuras de datos dentro del cálculo λ. Cada estructura de datos está definida por un conjunto finito de constructores.

- Booleanos: definidos por dos constructores sin argumentos `true` y `false`.

Estructuras de datos

Nos gustaría poder describir y usar estructuras de datos dentro del cálculo λ. Cada estructura de datos está definida por un conjunto finito de constructores.

- Booleanos: definidos por dos constructores sin argumentos `true` y `false`.
- Naturales: por dos constructores, uno unario, `S`, y otro sin argumentos, `Z`.
  - 0 se escribe como `Z`,
  - 1 se escribe como `S Z`,
  - 2 se escribe como `S (S Z)`
  - …

Estructuras de datos

Nos gustaría poder describir y usar estructuras de datos dentro del cálculo λ. Cada estructura de datos está definida por un conjunto finito de constructores.

- Booleanos: definidos por dos constructores sin argumentos `true` y `false`.
- Naturales: por dos constructores, uno unario, `S`, y otro sin argumentos, `Z`.
  - 0 se escribe como `Z`,
  - 1 se escribe como `S Z`,
  - 2 se escribe como `S (S Z)`
  - ...
- Listas: por dos constructores, uno para la lista vacía, `nil`, y otro para añadir un elemento a una lista `cons`.
  - `cons 3 (cons 2 (cons 1 nil))`

Contenido computacional

Una primera idea podría ser incluir directamente los constructores como constantes del lenguaje.

- `true`, `false`,
- `S`, `Z`.

Contenido computacional

Una primera idea podría ser incluir directamente los constructores como constantes del lenguaje.

- `true`, `false`,
- `S`, `Z`.

Pero esto no les da contenido computacional, no hay forma de calcular o definir funciones sobre ellos porque no pueden reducirse por β-reducción.

- `add (S (S Z)) (S Z)`,
- `and true false`.

Contenido computacional

Una primera idea podría ser incluir directamente los constructores como constantes del lenguaje.

- `true`, `false`,
- `S`, `Z`.

Pero esto no les da contenido computacional, no hay forma de calcular o definir funciones sobre ellos porque no pueden reducirse por β-reducción.

- `add (S (S Z)) (S Z)`,
- `and true false`.

Necesitamos una interpretación de estos constructores que se pueda usar para calcular.

Codificación de Church

La idea de la codificación de church es hacer depender a cada instancia de la estructura de datos de los constructores.

Codificación de Church

La idea de la codificación de church es hacer depender a cada instancia de la estructura de datos de los constructores. Por ejemplo, los números pueden ser funciones dependiendo de s y z

- $\lambda s.\lambda z.\ s\ (s\ z)$, sería el número 2, mientras que
- $\lambda s.\lambda z.\ s\ (s\ (s\ z))$, sería el número 3.

Codificación de Church

La idea de la codificación de church es hacer depender a cada instancia de la estructura de datos de los constructores. Por ejemplo, los números pueden ser funciones dependiendo de `s` y `z`

- $\lambda s.\lambda z.\ s\ (s\ z)$, sería el número 2, mientras que
- $\lambda s.\lambda z.\ s\ (s\ (s\ z))$, sería el número 3.

Los booleanos dependen de dos constructores

- $\lambda t.\lambda f.t$ es la constante `true`,
- $\lambda t.\lambda f.f$ es la constante `false`.

Codificación de Church II

Esto nos da una interpretación computacional de cada término, el natural `n` es una función de orden superior que toman una función y un argumento y aplica `n` veces la función sobre el argumento

$$3 \equiv (\lambda f.\lambda x.\ f\ (f\ (f\ x))).$$

Codificación de Church II

Esto nos da una interpretación computacional de cada término, el natural `n` es una función de orden superior que toman una función y un argumento y aplica `n` veces la función sobre el argumento

$$3 \equiv (\lambda f.\lambda x.\ f\ (f\ (f\ x))).$$

O los booleanos son funciones que toman dos argumentos y eligen uno de ellos

$$\mathrm{true} \equiv (\lambda x.\lambda y.x).$$

Usaremos estas propiedades para programar con ellos.