Programación Funcional Avanzada

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

Cálculo Lambda

Cálculo Lambda

Formalismo para representar funciones.

Gramática:

$$
\begin{array}{lll}
e ::= x & & \text{-- variables} \\
\quad | & (e1\ e2) & \text{-- aplicación} \\
\quad | & \lambda x.e & \text{-- lambda-abstracción}
\end{array}
$$

Cálculo Lambda

Formalismo para representar funciones.

Gramática:

$$
\begin{array}{lll}
e ::= x & & \text{-- variables} \\
\quad | & (e1\ e2) & \text{-- aplicación} \\
\quad | & \lambda x.e & \text{-- lambda-abstracción}
\end{array}
$$

Fue introducido por A. Church en los años 1930 con el objetivo de capturar el concepto de función computable.

En ese sentido el cálculo lambda es equivalente a las máquinas de Turing y otros formalismos.

Cálculo Lambda

Formalismo para representar funciones.

Gramática:

$$
\begin{array}{lll}
e ::= x & & \text{-- variables} \\
\quad \mid & (e1\ e2) & \text{-- aplicación} \\
\quad \mid & \lambda x.e & \text{-- lambda-abstracción}
\end{array}
$$

Fue introducido por A. Church en los años 1930 con el objetivo de capturar el concepto de función computable.

En ese sentido el cálculo lambda es equivalente a las máquinas de Turing y otros formalismos.

Construcciones como números y operaciones aritméticas se pueden agregar por conveniencia, pero no modifican su expresividad.

Variables

La construcción λx representa la **ligadura** (binding) de una variable. Corresponde a la declaración del parámetro formal de una función en un lenguaje de programación.

Las ocurrencia de una variable en una expresión es de dos tipos:

- **ligada** (bound)
  La variable está en el alcance de una ligadura.
  Por ejemplo, la ocurrencia x en $\lambda x.x$.
- **libre** (free)
  No hay una ligadura que la capture.

Ejemplos:

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

Variables libres de un término

$$
\begin{aligned}
FV\ (x) &= \{x\} \\
FV\ (e\ e') &= FV\ (e) \cup FV\ (e') \\
FV\ (\lambda x.e) &= FV\ (e) \backslash \{x\}
\end{aligned}
$$

Alfa-conversión

Regla de α-conversión: los nombres de las variables ligadas pueden ser cambiados, siempre y cuando no haya "captura de variables".

Alfa-conversión

Regla de α-conversión: los nombres de las variables ligadas pueden ser cambiados, siempre y cuando no haya "captura de variables".

Por ejemplo,

$\lambda x.x$ es α-convertible con $\lambda y.y$

Alfa-conversión

Regla de α-conversión: los nombres de las variables ligadas pueden ser cambiados, siempre y cuando no haya "captura de variables".

Por ejemplo,

$\lambda x.x$ es α-convertible con $\lambda y.y$

Sin embargo,

$\lambda x.\lambda y.x$ **no** es α-convertible con $\lambda y.\lambda y.y$

pero lo es con $\lambda z.\lambda y.z$

Beta-reducción

Una expresión de la forma

$$(\lambda x.e_1)\ e_2$$

se dice que es β-**redex** y puede ser β-**reducido** a

$$e_1\ [x := e_2]$$

en donde se sustituyen por e_2 las ocurrencias libres de la variable x en e_1 y, en caso de ser necesario, se realiza α-conversión para evitar la captura de variables libres en e_2.

Sustitución

$$
\begin{array}{rcl}
x\,[x := e] & = & e \\
y\,[x := e] & = & y \text{ si } y \not\equiv x \\
(e1\ e2)\,[x := e] & = & ((e1\,[x := e])\,(e2\,[x := e])) \\
(\lambda x.e')\,[x := e] & = & \lambda x.e' \\
(\lambda y.e')\,[x := e] & = & \lambda y.(e'\,[x := e]) \\
 & & \text{si} \quad y \not\equiv x \\
 & & \qquad y \notin FV\,(e) \\
(\lambda y.e')\,[x := e] & = & \lambda z.((e'\,[y := z])\,[x := e]) \\
 & & \text{si} \quad y \not\equiv x \\
 & & \qquad y \in FV\,(e) \\
 & & \qquad z \textit{ fresh}
\end{array}
$$

β-reducciones

Ejemplos de β-reducciones:

$(\lambda x.x)\ y$

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

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

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

β-reducciones

Ejemplos de β-reducciones:

$$(\lambda x.x)\ y \quad \rightarrow_\beta \quad y$$

$$(\lambda x.x)\ (\lambda x.x) \quad \rightarrow_\beta \quad \lambda x.x$$

$$(\lambda x.\lambda y.x)\ y \quad \rightarrow_\beta \quad \lambda z.y$$

$$(\lambda f.f\ x)\ (\lambda y.y) \quad \rightarrow_\beta \quad (\lambda y.y)\ x$$

β-reducciones

Ejemplos de β-reducciones:

$$(\lambda x.x)\ y \rightarrow_\beta y$$

$$(\lambda x.x)\ (\lambda x.x) \rightarrow_\beta \lambda x.x$$

$$(\lambda x.\lambda y.x)\ y \rightarrow_\beta \lambda z.y$$

$$(\lambda f.f\ x)\ (\lambda y.y) \rightarrow_\beta (\lambda y.y)\ x \rightarrow_\beta x$$

β-reducciones

Ejemplos de β-reducciones:

$$(\lambda x.x)\ y \rightarrow_\beta y$$

$$(\lambda x.x)\ (\lambda x.x) \rightarrow_\beta \lambda x.x$$

$$(\lambda x.\lambda y.x)\ y \rightarrow_\beta \lambda z.y$$

$$(\lambda f.f\ x)\ (\lambda y.y) \rightarrow_\beta (\lambda y.y)\ x \rightarrow_\beta x$$

Los términos sin β-redexes se dice que están en **forma normal**.

Reducciones divergentes

Hay términos que no reducen a una forma normal.

$$\omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$

$$\rightarrow_\beta$$

Reducciones divergentes

Hay términos que no reducen a una forma normal.

$$\omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$

$$\rightarrow_\beta\ (\lambda x.x\ x)\ (\lambda x.x\ x)$$

Reducciones divergentes

Hay términos que no reducen a una forma normal.

$$\omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$

$$\begin{aligned}
&\rightarrow_\beta (\lambda x.x\ x)\ (\lambda x.x\ x)\\
&\rightarrow_\beta (\lambda x.x\ x)\ (\lambda x.x\ x)\\
&\ldots
\end{aligned}$$

Múltiples redexes

Hay términos que tienen múltiples redexes.

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

redexes:

Múltiples redexes

Hay términos que tienen múltiples redexes.

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

redexes:

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

Múltiples redexes

Hay términos que tienen múltiples redexes.

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

redexes:

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

¿Por dónde empiezo a reducir?

Estrategias de evaluación

Call by value / evaluación estricta

La más común. Los argumentos se reducen tanto como se pueda antes de reducir la aplicación de una función, usualmente de izquierda a derecha.

Estrategias de evaluación

Call by value / evaluación estricta

La más común. Los argumentos se reducen tanto como se pueda antes de reducir la aplicación de una función, usualmente de izquierda a derecha.

Call by name

Las funciones se reducen antes que sus argumentos. Utilizado por algunos lenguajes de macros.

Estrategias de evaluación

Call by value / evaluación estricta

La más común. Los argumentos se reducen tanto como se pueda antes de reducir la aplicación de una función, usualmente de izquierda a derecha.

Call by name

Las funciones se reducen antes que sus argumentos. Utilizado por algunos lenguajes de macros.

Call by need / evaluación perezosa

Una versión optimizada de call by name: los argumentos se reducen sólo cuando se necesitan, pero se comparten si se usan múltiples veces.

Church-Rosser

Teorema (Church-Rosser)

Si un término e puede ser reducido a términos e_1 y e_2, entonces existe un término e_3 tal que tanto e_1 como e_2 reducen a él.

Church-Rosser

Teorema (Church-Rosser)

Si un término e puede ser reducido a términos e_1 y e_2, entonces existe un término e_3 tal que tanto e_1 como e_2 reducen a él.

Corolario

Cada término tiene como máximo una forma normal.

Church-Rosser

Teorema (Church-Rosser)

Si un término e puede ser reducido a términos e_1 y e_2, entonces existe un término e_3 tal que tanto e_1 como e_2 reducen a él.

Corolario

Cada término tiene como máximo una forma normal.

Teorema

Si un término tiene una forma normal, entonces la evaluación perezosa alcanza esa forma normal.

Haskell y el Cálculo Lambda

Haskell es una forma enriquecida de cálculo lambda.

Casi todas las construcciones de Haskell pueden ser traducidas al cálculo lambda.

El sistema de tipos funciona como una forma de restringir los términos lambda admisibles.

No terminación

En el cálculo lambda se pueden escribir términos cuya reducción no termina (diremos que su valor es $\perp$)

No terminación

En el cálculo lambda se pueden escribir términos cuya reducción no termina (diremos que su valor es $\perp$)

En Haskell la constante

$$undefined :: a$$

está definida para todo tipo y denota $\perp$.

No terminación

En el cálculo lambda se pueden escribir términos cuya reducción no termina (diremos que su valor es $\perp$)

En Haskell la constante

$$undefined :: a$$

está definida para todo tipo y denota $\perp$.

Valores de error también se interpretan como $\perp$.

- $error :: String \to a$ imprime un mensaje de error y corta el programa en lugar de divergir.
- división por cero

No terminación

En el cálculo lambda se pueden escribir términos cuya reducción no termina (diremos que su valor es $\perp$)

En Haskell la constante

$$undefined :: a$$

está definida para todo tipo y denota $\perp$.

Valores de error también se interpretan como $\perp$.

- $error :: String \to a$ imprime un mensaje de error y corta el programa en lugar de divergir.
- división por cero

Una función es estricta si $f \perp \equiv \perp$

Ejemplos de evaluación perezosa

1) $(\lambda x \to x)\ \mathit{True} \Rightarrow$
2) $(\lambda x \to x)\ \bot \Rightarrow$
3) $(\lambda x \to ())\ \bot \Rightarrow$
4) $(\lambda x \to \bot)\ () \Rightarrow$
5) $(\lambda x\ f \to f\ x)\ \bot \Rightarrow$
6) $\bot_1\ \bot_2 \Rightarrow$
7) $\mathit{length}\ (\mathit{map}\ \bot\ [1,2]) \Rightarrow$

Ejemplos de evaluación perezosa

$$
\begin{array}{lll}
1) & (\lambda x \to x)\ \mathit{True} & \Rightarrow\ \mathit{True} \\
2) & (\lambda x \to x)\ \bot & \Rightarrow\ \bot \\
3) & (\lambda x \to ())\ \bot & \Rightarrow\ () \\
4) & (\lambda x \to \bot)\ () & \Rightarrow\ \bot \\
5) & (\lambda x\ f \to f\ x)\ \bot & \Rightarrow \lambda f \to f\ \bot \\
6) & \bot_1\ \bot_2 & \Rightarrow\ \bot_1 \\
7) & \mathit{length}\ (\mathit{map}\ \bot\ [1,2]) & \Rightarrow\ 2
\end{array}
$$

Forzando la evaluación

Haskell tiene la siguiente función primitiva:

$$seq :: a \to b \to b$$

Forzando la evaluación

Haskell tiene la siguiente función primitiva:

$$seq :: a \to b \to b$$

La llamada:

$$seq\ x\ y$$

evalúa x antes de retornar y.

Forzando la evaluación

Haskell tiene la siguiente función primitiva:

$$seq :: a \to b \to b$$

La llamada:

$$seq\ x\ y$$

evalúa x antes de retornar y.

La reducción de x se realiza hasta una **forma normal débil**:

- lambda-abstracción
- literal
- aplicación de un constructor

Forzando la evaluación

Haskell tiene la siguiente función primitiva:

$$seq :: a \to b \to b$$

La llamada:

$$seq\ x\ y$$

evalúa x antes de retornar y.

La reducción de x se realiza hasta una **forma normal débil**:

- lambda-abstracción
- literal
- aplicación de un constructor

Puede ser usada para definir la **aplicación estricta**:

$$(\$!) :: (a \to b) \to a \to b$$
$$f\ \$!\ x = x\ `seq`\ f\ x$$

Ejemplos

$$
\begin{array}{lll}
1) & (\lambda x \to ())\ \$!\ \bot & \Rightarrow \\
2) & \mathit{seq}\ (\bot_1, \bot_2)\ () & \Rightarrow \\
3) & \mathit{snd}\ \$!\ (\bot_1, \bot_2) & \Rightarrow \\
4) & (\lambda x \to ())\ \$!\ (\lambda x \to \bot) & \Rightarrow \\
5) & \bot_1\ \$!\ \bot_2 & \Rightarrow \\
6) & \mathit{length}\ \$!\ \mathit{map}\ \bot\ [1,2] & \Rightarrow \\
7) & \mathit{seq}\ (\bot_1 + \bot_2)\ () & \Rightarrow \\
8) & \mathit{seq}\ (\mathit{foldr}\ \bot_1\ \bot_2)\ () & \Rightarrow \\
9) & \mathit{seq}\ (1 : \bot)\ () & \Rightarrow
\end{array}
$$

Ejemplos

$$
\begin{array}{lll}
1) & (\lambda x \to ())\ \$!\ \bot & \Rightarrow\ \bot \\
2) & \mathit{seq}\ (\bot_1, \bot_2)\ () & \Rightarrow\ () \\
3) & \mathit{snd}\ \$!\ (\bot_1, \bot_2) & \Rightarrow\ \bot_2 \\
4) & (\lambda x \to ())\ \$!\ (\lambda x \to \bot) & \Rightarrow\ () \\
5) & \bot_1\ \$!\ \bot_2 & \Rightarrow\ \bot_2 \\
6) & \mathit{length}\ \$!\ \mathit{map}\ \bot\ [1,2] & \Rightarrow\ 2 \\
7) & \mathit{seq}\ (\bot_1 + \bot_2)\ () & \Rightarrow\ \bot_1 \\
8) & \mathit{seq}\ (\mathit{foldr}\ \bot_1\ \bot_2)\ () & \Rightarrow\ () \\
9) & \mathit{seq}\ (1 : \bot)\ () & \Rightarrow\ ()
\end{array}
$$

Programando en Cálculo Lambda

Representación de booleanos

$$T = \lambda t.\lambda f.t$$
$$F = \lambda t.\lambda f.f$$

Representación de booleanos

$$T = \lambda t.\lambda f.t$$
$$F = \lambda t.\lambda f.f$$

$$NOT = \lambda a.\ \ a\ F\ T$$
$$AND = \lambda a.\lambda b.a\ b\ F$$
$$OR\ \ = \lambda a.\lambda b.a\ T\ b$$

Representación de pares

$$PAIR = \lambda f.\lambda s.\lambda b.b\ f\ s$$

$$FST\ = \lambda p.p\ T$$
$$SND\ = \lambda p.p\ F$$

Numerales de Church

$$\overline{0} = \lambda s.\lambda z.z$$
$$\overline{1} = \lambda s.\lambda z.s\ z$$
$$\overline{2} = \lambda s.\lambda z.s\ (s\ z)$$
$$\overline{3} = \lambda s.\lambda z.s\ (s\ (s\ z))$$
$$\ldots$$
$$\overline{n} = \lambda s.\lambda z.s^n\ z$$

Numerales de Church

$$\overline{0} = \lambda s.\lambda z.z$$
$$\overline{1} = \lambda s.\lambda z.s\ z$$
$$\overline{2} = \lambda s.\lambda z.s\ (s\ z)$$
$$\overline{3} = \lambda s.\lambda z.s\ (s\ (s\ z))$$
$$\ldots$$
$$\overline{n} = \lambda s.\lambda z.s^n\ z$$

$$Z = \lambda s.\lambda z.z$$
$$S = \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$$

Numerales de Church

$$\overline{0} = \lambda s.\lambda z.z$$
$$\overline{1} = \lambda s.\lambda z.s\ z$$
$$\overline{2} = \lambda s.\lambda z.s\ (s\ z)$$
$$\overline{3} = \lambda s.\lambda z.s\ (s\ (s\ z))$$
$$\ldots$$
$$\overline{n} = \lambda s.\lambda z.s^n\ z$$

$$Z = \lambda s.\lambda z.z$$
$$S = \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$$

$$PLUS = \lambda m.\lambda n.\lambda s.\lambda z.m\ s\ (n\ s\ z)$$

Numerales de Church

$$\overline{0} = \lambda s.\lambda z.z$$
$$\overline{1} = \lambda s.\lambda z.s\ z$$
$$\overline{2} = \lambda s.\lambda z.s\ (s\ z)$$
$$\overline{3} = \lambda s.\lambda z.s\ (s\ (s\ z))$$
$$\ldots$$
$$\overline{n} = \lambda s.\lambda z.s^n\ z$$

$$Z = \lambda s.\lambda z.z$$
$$S = \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$$

$$PLUS = \lambda m.\lambda n.\lambda s.\lambda z.m\ s\ (n\ s\ z)$$

$$ISZERO = \lambda n.n\ (\lambda x.F)\ T$$

Recursión

Término divergente:

$$\omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$

Generalización: Combinador de Punto Fijo

$$Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

Permite definiciones recursivas.

Recursión

Término divergente:

$$\omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$

Generalización: Combinador de Punto Fijo

$$Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

Permite definiciones recursivas.

$$\begin{aligned} Y\ g &\rightarrow_\beta (\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x)) \\ &\rightarrow_\beta g\ (\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x)) \\ &= g\ (Y\ g) \end{aligned}$$