Programando con Lambda-Cálculo

Mauro Jaskelioff

15/09/2017

Programando en λ-cálculo

- Vimos que el λ-cálculo es un cálculo con una sintaxis y semántica simple pero rica.
- Dijimos que con él se podrían representar todas las funciones computables.
- ¿Cómo programar con λ-cálculo?

Representaciones de tipos de datos

- Para programar, representamos los términos de los tipos de datos básicos (como naturales, booleanos, etc) con λ-expresiones.
  - Establecemos expresiones que representan los valores del tipo
    - Los constructores del tipo.
  - Establecemos expresiones que operan sobre el tipo.
    - Los eliminadores del tipo.
- Nos quedamos satisfechos cuando los valores y operadores del tipo cumplen con una especificación dada.
  - Como el λ-cálculo no tiene tipos, expresiones como $(not\ \underline{2})$ son válidas, pero no nos interesa como se comporten.
  - Escribiremos "definiciones" como $True \equiv (\lambda x\ y.\, x)$, pero esto es simplemente una abreviación expresada en nuestra metalenguaje.

Booleanos

- Queremos representar los valores True y False, y la operación $\mathit{ifthenelse}$
- Nuestra especificación es

$$\begin{aligned} &\mathit{ifthenelse}\ \mathit{True}\ \ P\ Q =_\beta P \\ &\mathit{ifthenelse}\ \mathit{False}\ P\ Q =_\beta Q \end{aligned}$$

- Por lo tanto

$$\begin{aligned} &\mathit{ifthenelse}\ \mathit{True}\ \ =_\beta \lambda p\ q.\, p \\ &\mathit{ifthenelse}\ \mathit{False} =_\beta \lambda p\ q.\, q \end{aligned}$$

- Una solución:

$$\begin{aligned} \mathit{True} &\equiv \lambda p\ q.\, p \\ \mathit{False} &\equiv \lambda p\ q.\, q \\ \mathit{ifthenelse} &\equiv \lambda x.\, x \end{aligned}$$

Mas operaciones sobre Booleanos

- Usando True, False e $\mathit{ifthenelse}$, definimos otras funciones (escribimos $\mathit{ifthenelse}\ P\ Q\ R$ como $\mathbf{if}\ P\ \mathbf{then}\ Q\ \mathbf{else}\ R$)

$$\begin{aligned}
\mathit{not} &\equiv \lambda x.\, \mathbf{if}\ x\ \mathbf{then}\ \mathit{False}\ \mathbf{else}\ \mathit{True}\\
&\equiv \lambda x.\, \mathit{ifthenelse}\ x\ \mathit{False}\quad \mathit{True}\\
&\equiv \lambda x.\, (\lambda x.\, x)\quad x\ (\lambda p\ q.\, q)\ (\lambda p\ q.\, p)\\
&\to_\beta \lambda x.\, x\ (\lambda p\ q.\, q)\ (\lambda p\ q.\, p)
\end{aligned}$$

$$\begin{aligned}
\mathit{not}\ \mathit{True} &\equiv (\lambda x.\, x\ (\lambda p\ q.\, q)\ (\lambda p\ q.\, p))\ (\lambda p\ q.\, p)\\
&\to_\beta (\lambda p\ q.\, p)\ (\lambda p\ q.\, q)\ (\lambda p\ q.\, p)\\
&\to_\beta (\lambda p\ q.\, q)\\
&\equiv \mathit{False}
\end{aligned}$$

$$\mathit{not}\ \mathit{False} \to_\beta \ldots \qquad (\mathit{Ejercicio}!)$$

- Otras funciones:

$$\begin{aligned}
\mathit{and} &\equiv \lambda x\ y.\, \mathbf{if}\ x\ \mathbf{then}\ y\ \mathbf{else}\ \mathit{False}\\
\mathit{or} &\equiv \lambda x\ y.\, \mathbf{if}\ x\ \mathbf{then}\ \mathit{True}\ \mathbf{else}\ y
\end{aligned}$$

Pares

- Queremos representar $pair$ y las operaciones fst y snd
- Nuestra especificación es

$$\begin{aligned} fst\ (pair\ P\ Q)\ &=_\beta P \\ snd\ (pair\ P\ Q) &=_\beta Q \end{aligned}$$

- Una solución:

$$\begin{aligned} pair &\equiv \lambda x\ y.\, \lambda b.\, \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y \\ fst &\equiv \lambda p.\, p\ True \\ snd &\equiv \lambda p.\, p\ False \end{aligned}$$

- Verifiquemos que $fst\ (pair\ x\ y) =_\beta x$:

$$\begin{aligned} fst\ (pair\ x\ y)\ &\equiv\ (\lambda p.\, p\ True)\ (pair\ x\ y) \\ &=_\beta (pair\ x\ y)\ True \equiv (\lambda b.\, \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y)\ True \\ &=_\beta \mathbf{if}\ True\ \mathbf{then}\ x\ \mathbf{else}\ y \\ &=_\beta x \end{aligned}$$

Receta para representaciones

1. Identificar **constructores**.
   - Como construir elementos.
2. Identificar **eliminadores**.
   - Como observar elementos.
3. Escribir **ecuaciones** con el comportamiento de los eliminadores sobre los constructores.
4. Definir λ-**términos** que satisfagan las ecuaciones.

Ejercicio

Dar una representación para el tipo $Either$ de Haskell:

$$\mathbf{data}\ Either\ a\ b = Left\ a \mid Right\ b$$

con el eliminador

$$\begin{aligned}
&either :: Either\ a\ b \to (a \to c) \to (b \to c) \to c \\
&either\ (Left\ a)\ \ \ f\ g = f\ a \\
&either\ (Right\ b)\ f\ g = g\ b
\end{aligned}$$

Representando Naturales

- Para representar números naturales podemos usar la técnica de Church.
- Los naturales son un tipo recursivo muy simple. En Haskell:

$$\textbf{data}\ Nat = Zero \mid Succ\ Nat$$

- Una forma estándar de eliminar tipos recursivos es el $fold$. En Haskell el fold para naturales es:

$$\begin{array}{ll}
foldn & :: Nat \to (a \to a) \to a \to a \\
foldn\ Zero\qquad\ \ s\ z & = z \\
foldn\ (Succ\ n)\ s\ z & = s\ (foldn\ n\ s\ z)
\end{array}$$

- Por lo tanto, para representar los naturales necesitamos definir:

$$\begin{array}{l}
Zero :: Nat \\
Succ :: Nat \to Nat \\
foldn :: Nat \to (a \to a) \to a \to a
\end{array}$$

Especificación de naturales

- Dado que

$$\begin{aligned} &foldn && :: Nat \to (a \to a) \to a \to a \\ &foldn\ Zero\ && s\ z = z \\ &foldn\ (Succ\ n)\ && s\ z = s\ (foldn\ n\ s\ z) \end{aligned}$$

  tomamos como especificación de los naturales

$$\begin{aligned} foldn\ Zero &=_\beta \lambda s\ z.\, z \\ foldn\ (Succ\ n) &=_\beta \lambda s\ z.\, s\ (foldn\ n\ s\ z) \end{aligned}$$

- Una solución se alcanza fijando $foldn = \lambda x.\, x$. Entonces:

$$\begin{aligned} Zero &=_\beta \lambda s\ z.\, z \\ Succ\ n &=_\beta \lambda s\ z.\, s\ (n\ s\ z) \end{aligned}$$

- A partir de las ecuaciones las definiciones son inmediatas:

$$Zero \equiv \lambda s\ z.\, z \qquad\qquad Succ \equiv \lambda n.\, \lambda s\ z.\, s\ (n\ s\ z)$$

Ejemplos

$$Zero \equiv \lambda s\ z.\, z$$
$$Succ \equiv \lambda n.\, \lambda s\ z.\, s\ (n\ s\ z)$$

$$\begin{aligned}
uno \equiv\ & Succ\ Zero \\
\equiv\ & (\lambda n.\, \lambda s\ z.\, s\ (n\ s\ z))\ Zero \\
=_\beta\ & \lambda s\ z.\, s\ (Zero\ s\ z) \\
\equiv\ & \lambda s\ z.\, s\ ((\lambda s\ z.\, z)\ s\ z) \\
=_\beta\ & \lambda s\ z.\, s\ z
\end{aligned}$$

$$\begin{aligned}
dos \equiv\ & Succ\ uno \\
\equiv\ & (\lambda n.\, \lambda s\ z.\, s\ (n\ s\ z))\ uno \\
=_\beta\ & \lambda s\ z.\, s\ (uno\ s\ z) \\
=_\beta\ & \lambda s\ z.\, s\ ((\lambda s\ z.\, s\ z)\ s\ z) \\
=_\beta\ & \lambda s\ z.\, s\ (s\ z)
\end{aligned}$$

$$tres =_\beta \lambda s\ z.\, s\ (s\ (s\ z))$$

Notación (metalenguaje)

- Queremos representar n aplicaciones de un término.
- En nuestro metalenguaje anotaremos

$$\begin{array}{lcl} F^0 \quad M & \equiv & M \\ F^{n+1} \; M & \equiv & F^n \; (FM) \end{array}$$

- Ejemplo:

$$\begin{array}{rcl} (\lambda x \; y.\, x)^3 \; z & \equiv & (\lambda x \; y.\, x)^2 \; ((\lambda x \; y.\, x) \; z) \\ & \equiv & (\lambda x \; y.\, x)^1 \; ((\lambda x \; y.\, x) \; ((\lambda x \; y.\, x) \; z)) \\ & \equiv & (\lambda x \; y.\, x)^0 \; ((\lambda x \; y.\, x) \; ((\lambda x \; y.\, x) \; ((\lambda x \; y.\, x) \; z))) \\ & \equiv & (\lambda x \; y.\, x) \; ((\lambda x \; y.\, x) \; ((\lambda x \; y.\, x) \; z)) \end{array}$$

- Notar que la notación sólo tiene sentido como parte del metalenguaje.

Numerales de Church

Definición

Para cada $n \in \mathbb{N}$*, el numeral de Church para* n *es un término* $\underline{n}$ *definido como*

$$\underline{n} \equiv \lambda f\ x.\ f^n\ x$$

- Notar que $\underline{0} \equiv \ \ False$. No importa, ya que no usamos tipos.
  - La especificación sólo dice que hacer en caso que los argumentos tengan la forma correcta.

Funciones sobre numerales de Church

- ¿Cómo definir la suma?
- una especificación de suma es la siguiente

$$\begin{aligned} suma\ \underline{n}\ \underline{0} &=_\beta \underline{n} \\ suma\ \underline{n}\ (Succ\ m) &=_\beta Succ\ (suma\ \underline{n}\ \underline{m}) \end{aligned}$$

- O equivalentemente:

$$suma\ \underline{n}\ \underline{m} =_\beta foldn\ \underline{m}\ Succ\ \underline{n}$$

- O sea que podemos definir

$$suma \equiv \ \lambda n\ m.\ m\ Succ\ n$$

La suma de $\underline{n}$ y $\underline{m}$ es aplicar la función sucesor m veces a $\underline{n}$.

Ejercicios

Ejercicio

Definir la multiplicación de naturales, para ello:

1. *Dar una especificación recursiva de la multiplicación.*
2. *Reescribir la especificación usando $foldn$.*
3. *Dar el término lambda correspondiente a la multiplicación.*

Ejercicio

Definir una función $isZero$, tal que

$$\begin{aligned} isZero\ \underline{0} &=_\beta True \\ isZero\ \underline{n+1} &=_\beta False \end{aligned}$$

Escribir la función $isZero$ usando $foldn$, y luego dar el λ-término.

Representando Listas

- Generalizamos los numerales de Church a listas.
- Las listas están dadas por el siguiente tipo de datos recursivo:

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

- Una forma estándar de consumir listas es con $foldr$. En Haskell:

$$\begin{array}{l} foldr :: List\ a \rightarrow (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow b \\ foldr\ Nil \qquad\qquad\quad c\ n = n \\ foldr\ (Cons\ x\ xs)\ c\ n = c\ x\ (foldr\ xs\ c\ n) \end{array}$$

- Por lo tanto, para representar listas necesitamos definir:

$$\begin{array}{ll} Nil & :: List\ a \\ Cons & :: a \rightarrow List\ a \rightarrow List\ a \\ foldr & :: List\ a \rightarrow (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow b \end{array}$$

Especificación de listas

- Dado que

$$\begin{aligned}
&\mathit{foldr}\ \mathit{Nil} && c\ n = n\\
&\mathit{foldr}\ (\mathit{Cons}\ x\ \mathit{xs})\ && c\ n = c\ x\ (\mathit{foldr}\ \mathit{xs}\ c\ n)
\end{aligned}$$

  tomamos como especificación de listas

$$\begin{aligned}
&\mathit{foldr}\ \mathit{Nil} && =_\beta \lambda c\ n.\, n\\
&\mathit{foldr}\ (\mathit{Cons}\ x\ \mathit{xs}) && =_\beta \lambda c\ n.\, c\ x\ (\mathit{foldr}\ \mathit{xs}\ c\ n)
\end{aligned}$$

- Una solución se alcanza fijando $\mathit{foldr} = \lambda x.\, x$. Entonces:

$$\begin{aligned}
&\mathit{Nil} && =_\beta \lambda c\ n.\, n\\
&\mathit{Cons}\ x\ \mathit{xs} && =_\beta \lambda c\ n.\, c\ x\ (\mathit{xs}\ c\ n)
\end{aligned}$$

- Por lo tanto definimos

$$\begin{aligned}
&\mathit{Nil} && \equiv\ \lambda c\ n.\, n\\
&\mathit{Cons} && \equiv\ \lambda x\ \mathit{xs}.\, \lambda c\ n.\, c\ x\ (\mathit{xs}\ c\ n)
\end{aligned}$$

Ejemplos

$$
\begin{aligned}
Nil &\equiv \lambda c\ n.\, n \\
Cons &\equiv \lambda x\ xs.\, \lambda c\ n.\, c\ x\ (xs\ c\ n)
\end{aligned}
$$

$$
\begin{aligned}
[3] \quad &\equiv \ Cons\ 3\ Nil \\
&=_\beta \lambda c\ n.\, c\ 3\ (Nil\ c\ n) \\
&\equiv \ \lambda c\ n.\, c\ 3\ ((\lambda c\ n.\, n)\ c\ n) \\
&=_\beta \lambda c\ n.\, c\ 3\ n
\end{aligned}
$$

$$
\begin{aligned}
[2,3] \quad &\equiv \ Cons\ 2\ (Cons\ 3\ Nil) \\
&=_\beta \lambda c\ n.\, c\ 2\ ((Cons\ 3\ Nil)\ c\ n) \\
&=_\beta \lambda c\ n.\, c\ 2\ ((\lambda c\ n.\, c\ 3\ n)\ c\ n) \\
&=_\beta \lambda c\ n.\, c\ 2\ (c\ 3\ n)
\end{aligned}
$$

$$
\begin{aligned}
[1,2,3] &\equiv \ Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil)) \\
&\equiv \ \lambda c\ n.\, c\ 1\ ((Cons\ 2\ (Cons\ 3\ Nil))\ c\ n) \\
&=_\beta \lambda c\ n.\, c\ 1\ ((\lambda c\ n.\, c\ 2\ (c\ 3\ n))\ c\ n) \\
&=_\beta \lambda c\ n.\, c\ 1\ (c\ 2\ (c\ 3\ n))
\end{aligned}
$$

Más ejemplos

- La función $length$, recursivamente

$$\begin{aligned} & length && :: List\ a \to Nat \\ & length\ Nil && = Zero \\ & length\ (Cons\ x\ xs) && = Succ\ (length\ xs) \end{aligned}$$

- Reescribimos como $foldr$:

$$length\ xs = foldr\ xs\ (\lambda x\ n.\, Succ\ n)\ Zero$$

- En λ-cálculo:

$$length \equiv \ \lambda xs.\, xs\ (\lambda x\ n.\, Succ\ n)\ Zero$$

Receta para representar tipos recursivos

Para representar un tipo de datos recursivo:

1. Identificar **constructores**
2. Escribir el **fold** correspondiente.
3. Tomar como **especificación** las ecuaciones del fold.
4. Definir el fold como la función identidad y **derivar** a partir de las ecuaciones los λ-términos correspondientes a los constructores.

Limitaciones de la representación con fold

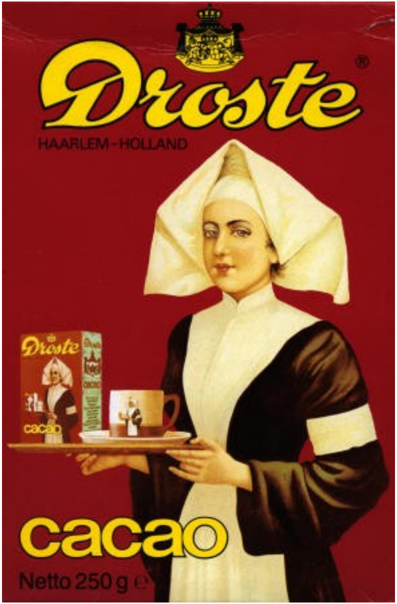
- La representación de tipos recursivos con fold trae algunas complicaciones.
- Algunas funciones muy comunes son difíciles de representar:
  - predecesor de los naturales
  - cola de una lista.
- Para poder definirlas es necesario utilizar tuplamiento
  - El resultado de la función es una tupla.
  - El resultado deseado está en una de las componentes.
- Por ejemplo, la función $pred'$ es un $foldn$.

$$\begin{aligned} pred'\ Zero &= (0,0) \\ pred'\ (Succ\ n) &= (snd\ (pred'\ n), Succ\ (snd\ (pred'\ n)) \end{aligned}$$

$$pred\ n = fst\ (pred'\ n)$$

$$pred' = \lambda n.\, foldn\ n\ (\lambda pn.\, (snd\ pn, Succ\ (snd\ pn)))\ (0,0)$$

Recursión general



Funciones recursivas

- ¿Cómo definir una función recursiva?
- Por ejemplo, queremos definir la función $fact$

$$fact \equiv \lambda n.\, \mathbf{if}\ (isZero\ n)\ \mathbf{then}\ \underline{1}\ \mathbf{else}\ prod\ n\ (fact\ (pred\ n))$$

  (suponemos ya definidas $prod$ (producto) y $pred$ $(predecesor)$)
- ¡Pero esto no es una definición válida! (¿Por qué?)
- Abstraemos la llamada recursiva problemática

$$B \equiv \lambda f\ n.\, \mathbf{if}\ (isZero\ n)\ \mathbf{then}\ \underline{1}\ \mathbf{else}\ prod\ n\ (f\ (pred\ n))$$

- La "definición" de más arriba se pude expresar como una ecuación:

$$fact =_{\beta} B\ fact$$

- Definir $fact$ es resolver esta ecuación en la incógnita $fact$.

Operador de punto fijo

- Para resolver en X una ecuación $X =_\beta B\ X$ se utilizan **operadores de punto fijo**.
- Un operador de punto fijo, es un término $\mathbf{F}$ tal que

$$\mathbf{F}\ B =_\beta B\ (\mathbf{F}\ B)$$

- Dado un operador de punto fijo $\mathbf{F}$ podemos definir $fact$:

$$\begin{aligned} fact \equiv &\ \ \mathbf{F}\ B \\ =_\beta &\ B\ (\mathbf{F}\ B) \\ \equiv &\ \ B\ fact \end{aligned}$$

- Es decir que $fact$ es:

$$fact \equiv \ \ \mathbf{F}\ (\lambda f\ n.\, \mathbf{if}\ (isZero\ n)\ \mathbf{then}\ \underline{1}\ \mathbf{else}\ prod\ n\ (f\ (pred\ n)))$$

El operador de puntos fijo $\mathbf{Y}$

- Considere el siguiente *combinador*:
  (un combinador es un término cerrado)

$$\mathbf{Y} \equiv \; \lambda x.\, (\lambda y.\, x \; (y \; y)) \; (\lambda y.\, x \; (y \; y))$$

Teorema

El combinador $\mathbf{Y}$ es un operador de punto fijo. Es decir, todo término X tiene un punto fijo dado por $(\mathbf{Y} \; X)$:

$$\mathbf{Y} \; X =_{\beta} X \; (\mathbf{Y} \; X)$$

- Nota: $\mathbf{Y}$ no es el único operador de punto fijo.

Resumen

- Representación de booleanos y pares.
- Representación de naturales.
- Representación de listas.
- Funciones recursivas y puntos fijos.

¡El λ-cálculo es un lenguaje de programación!

Referencias

- *Lambda-Calculus and Combinators*. J. R. Hindley and J. P. Seldin. Cambridge University Press (2008).

- *Theories of Programming Languages*. J. Reynolds (1998).
- *Types and Programming Languages*. B.C. Pierce (2002).