Cálculo Lambda

Dr. Víctor de la Cueva
vcueva@itesm.mx

Inicio

- Fue inventado por Alonzo Church en 1930 como un formalismo matemático para expresar operaciones (*computation*) de funciones, similar a la forma en la que la Máquina de Turing es un formalismo para expresar la operación (*computation*) de una computadora.
- El cálculo lambda se puede usar como modelo de un lenguaje funcional (puro) de la misma forma que la Máquina de Turing puede usarse como modelo para los lenguajes imperativos.
- Un gran resultado fue la demostración de que el cálculo lambda como una descripción de computación, es equivalente a la Máquina de Turing.
- Esto implica que el cálculo lambda, sin variables y sin asignación, con un estatuto if y recursión, es Turing completo (Turing complete).

Fuente foto: http://www.learn-math.info/historyDetail.htm?id=Church

Construcción

- El constructor básico del CL es la Abstracción Lambda:

$$(\lambda x. + 1\ x)$$

la cual puede ser interpretada en Scheme como

$$(\text{lambda}\ (x)\ (+ 1\ x))$$

- De la misma forma que en Scheme, el CL utiliza forma prefijo para sus funciones.
- La operación básica en CL es la aplicación de expresiones tales como la abstracción lambda:

$$(\lambda\ x\ .\ +\ 1\ x)\ 2$$

la cual representa la aplicación de la función al 2.

- El CL hace la aplicación por medio de una regla de reducción que permite que la x sea substituida por el 2 y quita la lambda: $(\lambda\ x\ .\ +\ 1\ x)\ 2 \Rightarrow (+\ 1\ 2) \Rightarrow 3$.

Sintaxis

- La sintaxis del CL es como sigue:

$$exp \rightarrow constant \mid variable \mid (exp\ exp) \mid (\lambda\ variable\ .\ exp)$$

- Las constantes en esta gramática son:
  - Los números: 1, 2, ..
  - Y ciertas funciones predefinidas como + y *
- La tercera regla representa la aplicación de funciones (f x).
- La cuarta regla es la abstracción lambda.
  - Sólo se permiten funciones de una sola variable, lo cual no es una gran restricción puesto que las expresiones lambda permiten el uso de expresiones de orden mayor, entonces se puede hacer currying.

Currying

- El nombre proviene del matemático y lógico Haskell B. Curry.
- El proceso conocido como currying permite que funciones de muchas variables sean interpretadas como funciones de orden superior de una sola variable que son regresadas al resto de las variables (funciones que regresan funciones).
- Es decir, las funciones de varias variables se obtienen por iteración de la aplicación de funciones de una sola variable. Esto es currying.
- A los lenguajes de programación que sólo permiten funciones de una sola variable (simple, no tupla) y si todas las funciones predefinidas son curried, se les conoce como completamente curry (*fully curried*).
- Haskell es completamente curry pero ML no, porque sus funciones aritméticas están definidas para recibir tuplas.

Fueten foto: https://wiki.haskell.org/Haskell_Brooks_Curry

Variables

- Las variables en CL no son como las variables en los lenguajes de programación.
  - No ocupan memoria porque el CL no tiene el concepto de memoria (como las máquinas de Turing).
- Las variables en CL corresponde a los parámetros de funciones.
- El conjunto de variables y el conjunto de constantes no es especificado por la gramática, por lo que se puede hablar, más correctamente, de muchos cálculos lambda:
  - Cada especificación del conjunto de constantes y variables define un CL en particular.
- CL sin constantes es llamado Cálculo Lambda Puro.

Paréntesis

- Los paréntesis son incluidos en las reglas de aplicación de funciones para evitar ambigüedad pero, por convención, pueden ser eliminados en caso de que no exista ambigüedad alguna.
- De acuerdo a la gramática debemos escribir:

  (λx. ((+ 1) x)) para (λx. + 1 x)

  (λx. ((+ 1) x) 2) para (λx. + 1 x) 2
- Pero como no existe ambigüedad, la segunda opción es permitida.
- Desde luego y como siempre, si hay dudas, usen paréntesis.

Acotada y libre

- La variable en la expresión (λx.*E*) se dice que está acotada (*bound*) por la lambda.
- El alcance (*scope*) de la variable x es el expresión E.
- La ocurrencia de una variable fuera del alcance de cualquier cota de una lambda se dice que es una ocurrencia libre.
- Una ocurrencia que no es libre se dice ocurrencia acotada.

  En la expresión (λx. + y x), y es libre y x es acotada

  Por lo que (λx. + y x) 2 ⇒ (+ y 2)

Diferentes lambdas

- Diferentes ocurrencias de las variables pueden estar acotadas por diferentes lambdas:

$$(\lambda x.\ +\ ((\lambda y.\ ((\lambda x.\ *\ x\ y)\ 2))\ x)\ y)$$

- Esta expresión se puede reducir como:

$(\lambda x.\ +\ ((\lambda y.\ ((\lambda x.\ *\ x\ y)\ 2))\ x)\ y) \Rightarrow$

$(\lambda x.\ +\ ((\lambda y.\ (*\ 2\ y))\ x)\ y) \Rightarrow$

$(\lambda x.\ +\ (*\ 2\ x)\ y)$

- Cuando se reducen expresiones que tienen múltiples cotas con el mismo nombre se debe tener mucho cuidado para no confundir los alcances.
- En algunas ocasiones se recomienda renombrar las variables.

Conversiones

- Debido a que el CL es tan general, se deben dar reglas muy precisas para la transformación de expresiones, tales como la sustitución de valores en variables acotadas.
- Estas reglas han mantenido sus nombres históricos y son conocidas como:
  - Conversión alfa (alpha-conversion o α-conversion)
  - Conversión beta (beta-conversion o β-conversion)
  - Conversión eta (eta-conversion o η-conversion)

Conversión beta

- El principal método de transformación para expresiones lambda es la sustitución o aplicación de función (e.g. ((λx. + x 1) 2) es equivalente a (+ 2 1)).
- Es exactamente como la llamada en un lenguaje de programación para una función definida por la abstracción lambda, con el 2 sustituyendo al valor del parámetro x.
- Históricamente, este proceso ha sido conocido como reducción beta (*beta-reduction*) en CL.
- Este proceso también se puede ver en revesa, donde (+ 2 1) se convierte en ((λx. + x 1) 2), el cual es llamado abstracción beta (*beta-abstraction*).
- El término para los dos procesos es conversión beta y establece la equivalencia de las dos expresiones.

Problema de la captura de nombre

- Formalmente, la conversión beta establece que ((λx.*E*) *F*) es equivalente a *E*[*F*/*x*], donde *E*[*F*/*x*] es *E* con todas las ocurrencias de x en E reemplazadas por F.
- Se debe tener mucho cuidado en las conversiones beta cuando F contiene nombre de variables que ocurren en E.

  ((λx.(λy. + x y)) y)

  Si se reduce con la y de afuera, que es libre, nos queda la reducción incorrecta:

  (λy. + y y)
- Esto es llamado el problema de la captura de nombre (*name capture problem*).
- Esto se soluciona renombrando la ocurrencia más interna de y:

  ((λx.(λy. + x y)) y) ⇒ ((λx.(λz. + x z)) y) ⇒ (λz. + y z)

Conversión alfa

- El cambio de nombre es otro proceso disponible en CL y es llamado alfa-conversion.
- Formalmente, (λx.*E*) es equivalente a (λy.*E*[*y*/*x*]), donde, *E*[*y*/*x*] es la expresión *E* con las ocurrencias libre de x reemplazadas por y.
- Desde luego que, si y es una variable que ya está en E podemos tener problemas similares a los ya discutidos.

Conversión eta

- Hay una conversión final que permite la eliminación de abstracciones lambda redundantes, la cual es llamada eta-conversion.
- Un ejemplo es la expresión: (λx. (+ 1 x)), como el 1 es *curried*, lo que está dentro del paréntesis se puede ver como la función (+ 1) aplicada a x, pero como x no está acotada en (+ 1), entonces da como resultado la función (+ 1 x), es decir, (λx. (+ 1 x)) 2, es lo mismo que (+ 1) 2.
- En general, (λx. (*E* x)) es equivalente a E, por eta-conversion, siempre que E no contenga ocurrencias libres de x.
  - Si el único propósito de una abstracción lambda es pasarle su argumento a otra función, entonces dicha abstracción lambda es redundante.
- Otro ejemplo, (λx. (λy.(+ x y))) ⇒ (λx. (+ x)) ⇒ +, es decir, la expresión inicial es otra notación para la función +.

Orden de la evaluación

- En particular, es posible distinguir entre evaluación de orden aplicativo (o paso por valor) de evaluación de orden normal (o paso por nombre o referencia).
- Ejemplo, ((λx. * x x) (+ 2 3))
  - Se puede usar orden aplicativo y reemplazar (+ 2 3) por las x y luego aplicar beta-reduction:

  ((λx. * x x) (+ 2 3)) ⇒ ((λx. * x x) 5) ⇒ (* 5 5) ⇒ 25
  - O aplicar beta-reduction primero y luego evaluar, dando una evaluación de orden normal:

  ((λx. * x x) (+ 2 3)) ⇒ (* (+ 2 3) (+ 2 3)) ⇒ (* 5 5) ⇒ 25
- La evaluación de orden normal es una especie de evaluación retardada (*delay evaluation*), debido a que la evaluación de expresiones es hecha después de la sustitución.

No siempre el mismo resultado

- No siempre dan el mismo resultado, por ejemplo, cuando la evaluación del parámetro da un resultado indefinido:
  - Si el valor de la expresión no depende del valor de su parámetro, el orden normal dará el resultado correcto mientras que el orden aplicativo dará un resultado indefinido.
- Por ejemplo, la expresión ((λx. x x) (λx. x x)):
  - La beta-reduction de esta expresión resulta en la misma expresión todo el tiempo, es decir, cae en un loop infinito tratando de reducir esta expresión.
  - Si usamos esta expresión como parámetro de una expresión constante ((λy. 2) ((λx. x x) (λx. x x))), el orden aplicativo dará un resultado indefinido mientras que el orden normal dará 2 puesto que el 2 no depende del valor del parámetro.
- Las funciones que regresan un valor aún cuando su parámetro es indefinido se llaman no estrictas (*nonstrict*), mientras que las funciones que son siempre indefinidas cuando sus parámetros son indefinidos se llaman estrictas (*strict*).

Reducción de orden normal

- La reducción de orden normal es muy significativa en CL debido a que, muchas expresiones pueden ser reducidas, usando orden normal, a una única forma normal que ya no puede ser reducida.
- Esto es una consecuencia del famoso teorema Church-Rosser, el cual establece que las secuencias de reducciones son esencialmente independientes del orden en el cual se realizan.

Definición

- La definición de una expresión lambda se puede renombrar con una variable para poder reducir la longitud, es decir, algo como un nombre:
  - Si hacemos F = λx. + x 1, podemos hacer F3, que es equivalente a (λx. + x 1)3
- Como es de esperarse, esto no es una asignación, simplemente es un renombramiento de la función completa
- En matemáticas fue muy utilizado al hacer definición de funciones:
  - $F(x) = x^2 + 2x - 1$, de aquí en adelante podríamos usar f(4) para referirnos al valor 23
- En el caso del CL, una vez dado un nombre a una función, este nombre se puede usar en lugar de la función
  - De la misma forma que las funciones lambda en Scheme se pueden nombrar y su nombre se usa en lugar de la función, por ejemplo (define f (lamba (x) (+ x 1))) y luego llamar a (f 4) que regresa 5 igual que ((lambda (x) (+ 1 x)) 4).

Recursión

- El CL puede modelar la definición de funciones recursivas exactamente en la misma forma en la que lo hemos hecho con Scheme.
- Por ejemplo, la definición de factorial fact = (λ*n*. (if (= *n* 0) 1 (* *n* (fact (- *n* 1)))))
- Podemos quitar la recursión como sigue:

  fact = (λ*F*. λ*n*. (if (= *n* 0) 1 (* *n* (*F* (- *n* 1))))) fact

- En otras palabras, el lado derecho puede ser visto como una abstracción lambda, la cual cuando se aplica a fact da como resultado fact nuevamente.
- Si escribimos la abstracción como *H* = (λ*F*. λ*n*. (if (= *n* 0) 1 (* *n* (*F* (- *n* 1))))), la ecuación se convierte en fact = *H* fact, y se dice que fact es un punto fijo de H.

Tarea de cálculo lambda

Referencias

- K.C. Louden and K. A. Lambert. Programming Languages: Principles and Practice. Cengage 3rd edition (2011).