Este documento son apuntes que tomé del libro 'The implementation of functional programming languages'[2] complementados con las teóricas[1] y bibliografía[3] sugerida de la materia.

Cálculo lambda

Se extiende el cálculo lambda puro

- Constantes: `true` , `false` , `0` , `nil`
- Funciones: `and` , `or` , `not` , `if` , `cons` , `head` , `tail` , `isnil`

Expresiones

```
<exp> ::= <constant>          -- built-in constants
        | <variable>          -- variable names
        | <exp> <exp>         -- applications
        | λ <variable>.<exp>  -- lambda abstractions
```

Semantica operacional

Variables libres y ligadas

Una variable **ocurre libre** en una expresión si no está **ligada** por una abstracción que la contiene.

Definicion de "libre":

- x occure libre en x
- x ocurre libre en (E F) <-> x ocurre libre en E o x ocurre libre en F
- x ocurre libre en (λy.E) <-> x ocurre libre en E y x ≠ y

Definicion de "ligada":

- x ocurre ligada en (E F) <-> x ocurre ligada en E o x ocurre ligada en F
- x ocurre ligada en (λy.E) <-> (x = y y x ocurre libre en E) o (x ocurre ligada en E)

Cada ocurrencia **individual** es ligada o libre, pero no ambas, sin embargo, una variable puede aparecer ligada y libre en la misma expresión. Por ejemplo: `(λx.x) x` .

Conversion beta (β)

Ejemplo: `(λx. x + 1) 4` Es la yuxtaposición de la abstración `(λx. x + 1)` y el argumento `4` .

Definición: El resultado de aplicar una abstracción a un argumento es la instancia del cuerpo de la abstracción que resulta de reemplazar las **ocurrencias libres** del paramétro de la abs. con (copias de) el argumento.

Ejemplo:

`(λx. x + 1) 4 -> 4 + 1` (β-reducción)

`4 + 1 <- (λx. x + 1) 4` (β-abstracción)

`4 + 1 <-> (λx. x + 1) 4` (β-conversión)

Data constructors

Se introducen los siguientes data-constructors: `CONS, HEAD, TAIL` donde:

- `HEAD (CONS a b) -> a`

- `TAIL (CONS a b) -> b`

Ejemplos:

- `CONS 1 (CONS 2 NIL)` (representa la lista [1, 2])
- `HEAD (CONS 1 (CONS 2 NIL)) -> 1`
- `TAIL (CONS 1 (CONS 2 NIL)) -> (CONS 2 NIL)`

Estos constructores pueden ser representados como abstracciones:

- `CONS = (λa. λb. λf. f a b)`
- `HEAD = (λx. x (λa. λb. a))`
- `TAIL = (λx. x (λa. λb. b))`

`CONS 1 NIL` es equivalente a `(λa. λb. λf. f a b) 1 NIL -> (λ.f f 1 NIL)`

`HEAD (CONS 1 NIL)` es equivalente a `(λx. x (λa. λb. a)) (λ.f 1 NIL) -> (λ.f f 1 NIL) (λa. λb. a) -> (λa. λb. a) 1 NIL -> (λb. 1) NIL -> 1`

`TAIL (CONS 1 NIL)` es equivalente a `(λx. x (λa. λb. b)) (λ.f 1 NIL) -> (λ.f f 1 NIL) (λa. λb. b) -> (λa. λb. b) 1 NIL -> (λb. b) NIL -> NIL`

Conversion alpha (α)

La conversión alpha es una operación que cambia los nombres de las **variables ligadas** en una expresión.

Ejemplo:

`(λx. x+1) <-> (λy. y+1)` (α-conversión)

El nuevo nombre de la variable ligada no debe ser igual a ninguna variable libre en el cuerpo de la abstracción.

Conversion eta (η)

La conversión eta es una operación que elimina abstracciones redundantes. En forma general, si `x` no ocurre libre en `F`, entonces:

`(λx. F x) <-> F` (η-conversión)

F puede ser cualquier función.

Como probar la interconvertibilidad de dos expresiones

Vamos a demostrar que se puede probar la interconvertiblidad de dos expresiones si aplicandolas al mismo argumento se obtiene el mismo resultado.

Dadas dos expresiones `F1` y `F1` y una variable `w` tal que no aparece libre en `F1` o `F1`, si

`F1 w <-> E` y `F2 w <-> E` entonces `F1 <-> F2`.

Prueba:

- `F1 <-> (λw. F1 w)` (η-conversión)
- `<-> (λw. E)` (Por hipótesis)
- `<-> (λw. F2 w)`

- `<-> F2` (η-conversión)

Sustitución

La sustitución es una operación que reemplaza todas las ocurrencias libres de una variable en una expresión por otra expresión.

Notación: `E[M/x]` significa que se reemplazan todas las ocurrencias libres de `x` en `E` por `M`. Y se define de la sig. manera:

- `x[M/x] -> M`
- `c[M/x] -> c` (c es una constante o una variable distinta a x)
- `(E F)[M/x] -> E[M/x] F[M/x]`
- `(λx.E)[M/x] -> λx.E` (no podemos sustituir variables ligadas)
- `(λy.E)[M/x] -> (λy.E[M/x])` si **x** no ocurre libre en E o **y** no ocurre libre en M. (**y** es una variable distinta a x)
- `(λy.E)[M/x] -> λz.(E[y/z])[M/x]` si no se cumple lo anterior. z es una variable fresca (variable que no aparece libre en E o M)

Definicion de todas las reducciones

- `(λx.E) M <-> E[M/x]` (β-conversión)
- `(λx.E) <-> (λy.E[y/x])` (α-conversión)
- `(λx.E x) <-> E` (η-conversión)

Semantica operacional del calculo lambda extendido a naturales y booleanos

La siguiente es la semantica operacional de λ^{BN}(lambda calculus with booleans and naturals descrito en la teórica de la materia de la cátedra[1]) esta extension es distinta a la que se describe en este libro[2], pero en algunos aspectos son iguales.

- `(λx.M) N -> M[N/x]` (β-reducción)
- `if true then M else N -> M` (if_t)
- `if false then M else N -> N` (if_f)

Tambien existen **reglas de congruencia** que permiten reducir subexpresiones de una expresion.
Si `M -> N` entonces:

- `M O -> N O` (app_l)
- `O M -> O N` (app_r)
- `λx.M -> λx.N` (fun)
- `if M then N else O -> if N then O else O` (if_c)

Orden de reducción

Una expresion está en **forma normal** si no contiene ninguna subexpresion que pueda ser reducida (*redexes*).

La evaluacion de una expresion es una secuencia de reducciones hasta que la expresion resultante quede en forma normal.

Una expresion se puede reducir de diferentes maneras. Algunas expresiones pueden no tener forma normal. Pero podemos demostrar que una expresion tiene una única forma normal.

Primer teorema de Church-Rosser (CRT 1)

> *Si `E1 <-> E2` entonces existe una expresión `E` tal que:*
>
> - *`E1 -> E` y `E2 -> E`*

Es equivalente a:

> *Si `E1 -> E2` y `E1 -> E3` entonces existe una expresión `E` tal que:*
>
> - *`E2 -> E` y `E3 -> E`*

Y se conoce como **propiedad del diamante**

Corolario:
Una expresion no puede ser reducida a dos formas normales distintas (que no sean α-convertibles).

Este corolario nos dice que todas las secuencias de reducciones finitas llegan al mismo resultado.

Segundo Teorema de Church-Rosser (CRT 2)

El CRT2 tiene que ver con una estrategia de reduccion llamada **estrategia de reduccion de orden normal**. La cual consiste en reducir primero el redex mas externo de mas a la izquierda.

Teorema:

> *Si `E1 -> E2` y `E2` está en forma normal, entonces existe una secuencia de reducciones de orden normal desde `E1` hasta `E2`.*

*Nota: En la teórica[1] de la materia, esta estrategia es call-by-name. Y call-by-name débil es normal order reduction pero con la restricción de que las expresiones que son abstracciones nunca se reducen, entonces, traduciendo los nombres de las teóricas: **call-by-name**: normal order reduction **call-by-name débil**: call-by-name*

Estrategias de reducción

La noción de estrategia de reducción permite definir el orden en el cual se debe reducir un término.

Reducción débil

Una estrategia de reduccion es **débil** si no reduce nunca el cuerpo de una abstracción. Una estrategia debil no optimiza programas, solo los ejecuta.

Call-by-name (CBN)

Call-by-name reduce siempre el redex más a la izquierda. En caso de ser call-by-name débil, reducirá el redex de mas a la izquierda que no esté bajo un λ (abstracción).

TEOREMA DE ESTANDARIZACIÓN (CRT 2)

El segundo teorema de Church-Rosser nos dice que si una expresion tiene una forma normal, entonces la estrategia de reducción call-by-name llegará a ella.

Esta estrategia no evalúa los argumentos de una función hasta que estos sean utilizados por el cuerpo de la función. Esto puede ser eficiente o no, dependiendo del caso:

- Si el argumento es computacionalmente costoso y no es utilizado, entonces call-by-name es eficiente ya que nunca se va a evaluar.
- Si el argumento es computacionalmente costoso y es utilizado mas de una vez en el cuerpo de la función, entonces call-by-name es ineficiente ya que se va a evaluar mas de una vez.

Se puede optimizar el último caso utilizando **reducción lazy**.

Call-by-value (CBV)

Valores

Si un término M está en forma normal y no tiene variables libres (FV(M) = ∅), entonces M es un **valor**.

La estrategia call-by-value consiste en evaluar siempre los argumentos antes de pasarlos a la función.
`(λx.M) V` reduce solamente cuando V está en forma normal.
En caso de ser call-by-value débil, V debe ser un valor.

Funciones recursivas

Se puede implementar la recursión en el calculo lambda extendiendolo con un nuevo símbolo.

Punto fijo

Supongamos que queremos definir recursivamente la función factorial

`F = λn. if n = 0 then 1 else n * F(n - 1)`

Obviamente no podemos definir F en términos de F ya que la sintaxis del cálculo lambda no lo permite (No existen las asignaciones a variables y las funciones tamnpoco tienen nombre).
Definimos F de la siguiente manera usando β-abstracción:

```
F = (λf. (λn. if n = 0 then 1 else n * f(n - 1))) F
F  = (λf. (... f ...)) F
```

Luego vemos que F es el punto fijo (fixed-point) de la función `λf. (...f...)`. Si definimos otro símbolo (μ) que represente el **punto fijo**, conseguiremos definir F :)

> *`μf. M` es el punto fijo de `λf. M` `μf. M` se comporta como `(λf. M) (μf. M)`*

Por lo tanto `F = μf. (...f...)` `F = μf. (λn. if n = 0 then 1 else n * f(n - 1))`

Y-combinator

Se puede definir el punto fijo sin necesidad extender el calculo lambda sin extenderlo con el símbolo μ definiendo una función de orden superior llamada **paradoxical combinator**

`Y = λf. (λx. f (x x)) (λx. f (x x))`

La cual cumple las propiedades de μ.

`Y F = (λx. F (x x)) (λx. F (x x)) = F (Y F)`

Tipos simples

Vamos a restrinigir las clases de conjuntos que se pueden utilizar como dominios de funciones. A estos conjuntos los llamamos **tipos**.

Definamos la gramatica para tipos simples del calculo lambda extendido con booleanos:
Los tipos los definimos inductivamente como:^

- `Bool` es un tipo
- Si `τ` y `σ` son tipos, entonces `τ -> σ` es un tipo

Gramática:

```
τ ::= Bool | Nat | τ -> τ
M := x | λx.M | M M
    | true | false | if M then M else M
    | Zero | Succ(M) | Pred(M) | IsZero(M)
```

Reglas de tipado

Queremos definir la relación `Γ ⊢ M : τ` que significa que " `M` tiene tipo `τ` bajo el contexto `Γ` "

El contexto nos da tipos para variables libres dentro de la expresión M.

Las reglas de tipado se definen inductivamente como:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\, ax_v$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma . M \ : \sigma \to \tau} \to_i$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \to_e$$

$$\frac{}{\Gamma \vdash \text{true} : Bool}\, ax_{true} \quad \frac{}{\Gamma \vdash \text{false} : Bool}\, ax_{false}$$

$$\frac{\Gamma \vdash M : Bool \quad \Gamma \vdash P : \sigma \quad \Gamma \vdash Q : \sigma}{\Gamma \vdash \text{if M then P else Q} : \sigma}\text{if}$$

$$\frac{}{\Gamma \vdash \text{Zero} : \text{Nat}}\text{zero} \quad \frac{\Gamma \vdash M : \text{Nat}}{\Gamma \vdash \text{isZero(M)} : Bool}\text{isZero}$$

$$\frac{\Gamma \vdash \text{M} : \text{Nat}}{\Gamma \vdash \text{pred(M)} : \text{Nat}}\text{pred} \quad \frac{\Gamma \vdash \text{M} : \text{Nat}}{\Gamma \vdash \text{succ(M)} : \text{Nat}}\text{succ}$$

$$\frac{\Gamma, x : \tau \vdash M : \tau}{\Gamma \vdash \mu x . M : \tau}\text{fix}$$

Teorema de preservación de tipos (subject reduction)

> *Si* `Γ ⊢ M : τ` *y* `M -> N` *entonces* `Γ ⊢ N : τ`

Si deducimos el tipo `τ` para una expresión M y luego al ejecutarla obtenemos la expresion N, entonces N también tiene tipo `τ` .

Teorema de Tait (strong normalization)

> *La relación de reducción (β-reduction) es fuertemente normalizable en el calculo lambda con tipos simples.*

Este teorema afirma que todos los términos bien tipados en este sistema son fuertemente normalizables, garantizando que todos los terminos a ser ejecutados terminan.

Término fuertemente normalizable

Un término M es fuertemente normalizable si no existe una secuencia infinita de reducciones N0 -> N1 -> ..., tal que M -> N0 -> N1 -> ...

Polimorfismo

El polimorfismo es una técnica que permite definir funciones que pueden ser aplicadas a diferentes tipos de datos.

Sabemos que a la función `λx.x` se le deriva el tipo `τ -> τ`, cualquiera sea el tipo `τ`. Entonces podemos introducir una **variable de tipo** X y atribuirle a `λx:X. x` el tipo:

$$\forall X.\,; X \to X$$

Agregando una regla de tipado para la cual si M tiene tipo $\forall X.\,; \tau$, entonces M tiene tipo $\tau[\sigma/X]$ para cualquier tipo σ.

Se presentan dos sistemas para implementar el polimorfismo: **Sistema F** y **Polimorfismo let**.

- Sistema F : Mas general, pero no es **dirigido por sintáxis**. Es decir, dado un término M, podemos aplicar distintas reglas de tipado. Inferir tipos en este sistema es **indecidible** (no existe un algoritmo para inferir tipos para cualquier término).
- Polimorfismo let : Menos general, pero es dirigido por sintáxis y **decidible**.

Sistema F

Se agregan las siguientes reglas de tipado:

$$\frac{\Gamma \vdash M : \tau \quad X \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash M : \forall X.\,\tau}\forall_i \qquad \frac{\Gamma \vdash M : \forall X.\,\tau}{\Gamma \vdash M : \tau[\sigma/X]}\forall_e$$

En la regla $\forall_i$ se pide que la variable de tipo X no aparezca en las variables libres del contexto Γ. Por ejemplo si en el contexto tenemos que $\Gamma = x : X$, entonces no podemos usar la variable X, en cambio si está siendo ligada por otro cuantificador, si podemos. Por ejemplo $\Gamma = x : \forall X.\,\tau$

Si $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, entonces $\mathrm{FV}(\Gamma) = \mathrm{FV}(\tau_1) \cup \ldots \cup \mathrm{FV}(\tau_n)$

Se define $\mathrm{FV}(\tau)$ de manera inductiva sobre el conjunto de tipos: $\tau ::= X, \mathit{Bool}, \tau \to \tau, \forall X.\,\tau$.

- $\mathrm{FV}(X) = X$
- $\mathrm{FV}(\mathit{Bool}) = \emptyset$
- $\mathrm{FV}(\tau \to \sigma) = \mathrm{FV}(\tau) \cup \mathrm{FV}(\sigma)$
- $\mathrm{FV}(\forall X.\,\tau) = \mathrm{FV}(\tau) - X$

Polimorfismo let

Se extiende el calculo lambda con un nuevo término equivalente a (λx.M) N : `let x = N in M`

Y con una nueva regla de semántica operacional (reduce igual que la expresion (λx.M) N): `let x = N in M -> M[N/x]`

Y con una nueva regla de tipado:

$$\frac{\Gamma \vdash N : \tau \quad \Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \mathrm{let}\ x = N\ \mathrm{in}\ M : \sigma}\mathrm{let}$$

Solo se permiten $\forall$ en la variable ligada por el término **let**. Las variables ligadas por abstracciones no permiten tipos polimórficos.
Los tipos ligados por cuantificadores ahora se llaman **esquemas de tipos**.

Esquemas de tipos

Un esquema de tipo tiene forma $\forall X_1 \ldots \forall X_n.\,[\tau]$
Con $n \geq 0$ y τ un tipo. Por ejemplo, $[\tau]$ es un esquema de tipo formado por el tipo τ donde ninguna variable está cuantificada.

Se define la gramática de tipos de la siguiente manera:

```
τ ::= X | Bool | τ -> τ
e ::= [τ] | ∀X.e
```

Sistema dirigido por sintaxis de Hindley-Milner

Se define un sistema de tipos polimorficos en el que cada término tiene una sola regla para derivar su tipo (o sea digamos, dirigido por sintáxis)

Relación de orden entre esquemas de tipos

Se define una relación de orden entre esquemas de tipos ($\preceq$) de la siguiente manera:

$$e \preceq \forall X.\, e \text{ ;si; } X \notin \mathrm{FV}(e)$$

$$\forall X.\, e \preceq eX := \tau$$

Cierre de un esquema de tipo respecto a un contexto

El cierre de un esquema de tipo respecto a un contexto es un esquema de tipo que cuantifica todas las variables libres de un tipo que no están en el contexto.

Sea Γ un contexto, τ un tipo y $\vec{X} = \mathrm{FV}(\tau) - \mathrm{FV}(\Gamma)$.
Definimos el cierre de τ respecto a Γ como:

$$\vec{\Gamma}(\tau) = \forall \vec{X}.\, [\tau]$$

Por ejemplo, el cierre de $X \to Y \to Z$ con respecto a $\Gamma = x : X, w : W$
es : $\vec{\Gamma}(\tau) = \forall (X, Y, Z - X, W).\, [\tau] = \forall Y.\, \forall Z.\, [X \to Y \to Z]$

Reglas de tipado

$$\frac{e \preceq e'}{\Gamma, x : e \vdash x : e'} ax_v$$

$$\frac{\Gamma, x : [\tau] \vdash M : [\sigma]}{\Gamma \vdash \lambda x : [\tau].\, M : [\tau \to \sigma]} \to_i \quad \frac{\Gamma \vdash M : [\sigma \to \tau] \quad \Gamma \vdash N : [\tau]}{\Gamma \vdash MN : [\sigma]} \to_e$$

$$\frac{}{\Gamma \vdash \mathrm{true} : [\texttt{Bool}]} ax_t \quad \frac{}{\Gamma \vdash \mathrm{false} : [\texttt{Bool}]} ax_f \quad \frac{\Gamma \vdash M : [\texttt{Bool}] \quad \Gamma \vdash P : [\tau] \quad \Gamma \vdash Q : [\tau]}{\Gamma \vdash \mathsf{if}\ M; \mathsf{then}\ P; \mathsf{else}\ Q} [\tau]\mathsf{if}$$

$$\frac{\Gamma \vdash N : [\tau] \quad \Gamma, x : \vec{\Gamma} \vdash M : [\sigma]}{\Gamma \vdash \mathsf{let}\ x = N\ \mathsf{in}\ M : [\sigma]} \mathsf{let}$$

Interpretación

Un intérprete del calculo lambda es un programa que toma una expresión y devuelve el valor al que reduce (su forma normal).

Interpretación en Call-by-name (CBN)

Por ejemplo: para el término `(λx. x * x) (2+2)`
El intérprete debería guardar `x = 2 + 2` en una estructura anexa llamada **contexto** y evaluar `x * x` en dicho contexto.

El intérprete evalúa términos con variables libres y cuando queremos evaluar una variable en si, la buscamos en el contexto.

Si en el contexto encontramos `x = M` donde M no es un valor, entonces deberíamos encontrar tambien el contexto en el que `M` debe ser evaluado.

Contexto

Un contexto es una función que mapea variables a términos. Escribimos un contexto como una **lista de pares** `x1 = M1, x2 = M2, ... xn = Mn`.
Una variable `x` puede aparecer varias veces en un contexto, en ese caso, `x` toma la primer aparicion desde la derecha.
Si `Γ` es un contexto y `x = t` un par, notamos `Γ , x = t` como el contexto `Γ` extendido con `x = t`.

Thunk

Un thunk es un par $\langle M, \Gamma \rangle$ formado por un término M y un contexto de evaluación Γ.

Cierre

Cuando queremos evaluar una abstracción `(λx.M)` en un contexto, el resultado debe ser el término `M` pero **cerrado** (la variable `x` debe estar ligada en el contexto).
Un cierre es un valor formado por una variable x, un término M y un contexto Γ y se nota: $\langle x, M, \Gamma \rangle$

Relación -> en CBN

Definimos la relación $\Gamma \vdash M \hookrightarrow V$ que se lee como: **el término M se interpreta como el valor V en el contexto Γ**.

Semántica denotacional (TO-DO)

La semantica operacional es una forma de definir el significado de una función de manera **dinámica**, como una serie de operaciones que se realizan para obtener un resultado, mas parecido a un algoritmo.

La semántica denotacional es una forma de definir el significado de una función de manera **estática**, como un conjunto de asociaciones entre argumentos y sus valores correspondientes. Sirve para relacionar las funciones del cálculo lambda con la idea de funciones abstractas.

Fuentes

[1] - Apunte de las clases de cálculo lambda, Alejandro Díaz-Caro (Jano).

[2] - The Implementation of Functional Programming Languages, Simon L. Peyton Jones.

[3] - Introduction to the theory of programming languages, G.Dowek y J.-J. Lévy.