CÁLCULO LAMBDA

Motivación: notación para no necesitar nombrar funciones. Por ejemplo, $x + y$ puede ser:

$$f(x) = x + y$$
$$g(y) = x + y$$
$$h(x, y) = x + y$$

La notación lambda permite expresar sin dar nombre:

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

e incluso

$$\lambda x.\lambda y.x + y$$
$$\lambda y.\lambda x.x + y$$

En algunos contextos, los matemáticos usan notaciones similares: En integrales, $\int x + y\,dx$, dx enfatiza que x varía e y está fija. En $\sum_{i \in 0.,10} i + j$, i varía y j está fija.

El cálculo lambda, en principio es una notación para no tener que nombrar funciones. Pero además, el cálculo lambda puro, sólo contiene variables, aplicaciones y la notación lambda (llamada abstracción).

$\langle \mathsf{expr} \rangle$	$::=$		término lambda, o expresión
		$\langle \mathsf{var} \rangle$	variable
	$\mid$	$\langle \mathsf{expr} \rangle \langle \mathsf{expr} \rangle$	aplicación, el primero es el operador y el segundo el operando
	$\mid$	$\lambda \langle \mathsf{var} \rangle . \langle \mathsf{expr} \rangle$	abstracción o expresión lambda

La aplicación asocia a izquierda. En $\lambda v.e$, la primer ocurrencia de v es ligadora y su alcance es e. Por ejemplo, $\lambda x.(\lambda y.xyx)x$ es lo mismo que $\lambda x.((\lambda y.((xy)x))x)$.

Puesto que λ se comporta como un cuantificador, se hacen exactamente las mismas definiciones de ocurrencia ligada, ocurrencia libre y variable libre. El conjunto de variables libres se define también por inducción en la estructura de los términos:

$$FV(v) = \{v\}$$
$$FV(e_0 e_1) = FV(e_0) \cup FV(e_1)$$
$$FV(\lambda v.e) = FV(e) - \{v\}$$

También se define sustitución:

$$\begin{aligned}\Delta =& \langle \mathsf{var} \rangle \to \langle \mathsf{expr} \rangle \\ _/_ \in& \langle \mathsf{expr} \rangle \times \Delta \to \langle \mathsf{expr} \rangle \\ v/\delta =& \delta v \\ (e_0 e_1)/\delta =& (e_0/\delta)(e_1/\delta) \\ (\lambda v.e)/\delta =& \lambda v'.(e/[\delta | v : v']) \\ & \text{donde } v' \notin \bigcup_{w \in FV(e)-\{v\}} FV(\delta w)\end{aligned}$$

Como veremos, la sustitución en el cálculo lambda es fundamental: permite definir la semántica operacional.

Propiedad 1.
1. *si para todo* $w \in FV(e)$, $\delta w = \delta' w$ *entonces* $(e/\delta) = (e/\delta')$
2. *sea* i *la sustitución identidad, entonces* $e/i = e$.
3. $FV(e/\delta) = \bigcup_{w \in FV(e)} FV(\delta w)$

Escribimos $e/v \to e'$ en vez de $e/[i|v : e']$ y $e/v_1 \to e_1, \ldots, v_n \to e_n$ en vez de $e/[i|v_1 : e_1| \ldots |v_n : e_n]$.

Renombre (α). La operación de cambiar una ocurrencia de la expresión lambda $\lambda v.e$ por $\lambda v'.(e/v \to v')$ donde $v' \notin FV(e) - \{v\}$ se llama renombre o cambio de variable ligada. Si e' se obtiene a partir de e por cero o más renombres de ocurrencias de subfrases, se dice que e' se obtiene a partir de e por renombre. También se dice que "e α-convierte a e'". Se puede ver que ésta es una relación de equivalencia. Se dice que e y e' son α-equivalentes y se escribe $e \equiv e'$. Se supone que renombre debe preservar la semántica (aunque hubo algunas excepciones en los primeros lenguajes funcionales).

Contracción (β). Una expresión de la forma $(\lambda v.e)e'$ se llama redex (plural redices, a veces mal dicho rédexes). Es la aplicación de una función $(\lambda v.e)$ a su argumento e'. Debe calcularse reemplazando las ocurrencias libres de v en e por e', es decir $(e/v \to e')$.

Cuando en una expresión e_0 se reemplaza una ocurrencia de un redex $(\lambda v.e)e'$ por $(e/v \to e')$ y luego de cero o más renombres se obtiene e_1, se dice que "e_0 β-contrae a e_1" y se escribe $e_0 \to e_1$. A pesar que utilizamos el término contracción, no necesariamente e_1 va a ser más pequeña que e_0. Por ejemplo $e_0 = (\lambda x.xxxx)(\lambda y.\lambda z.yz)$ β-contrae a $(\lambda y.\lambda z.yz)(\lambda y.\lambda z.yz)(\lambda y.\lambda z.yz)(\lambda y.\lambda z.yz)$ que es más grande.

Ejemplos:

$$\begin{array}{ll} (\lambda x.y)(\lambda z.z) & \to y \\ (\lambda x.x)(\lambda z.z) & \to (\lambda z.z) \\ (\lambda x.xx)(\lambda z.z) & \to (\lambda z.z)(\lambda z.z) \to (\lambda z.z) \\ (\lambda x.(\lambda y.yx)z)(zw) & \to (\lambda x.zx)(zw) \to z(zw) \\ (\lambda x.(\lambda y.yx)z)(zw) & \to (\lambda y.y(zw))z \to z(zw) \end{array}$$

Podemos comprobar que las últimas expresiones de cada línea no contienen redices, por lo tanto no se pueden contraer más. Una expresión sin redices se llama *forma normal*. Los últimos dos ejemplos comienzan con la misma expresión, se diferencian, pero cuando llegan a la forma normal dan el mismo resultado.

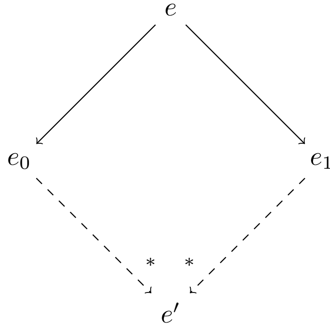
Utilizando la notación de la semántica de transiciones para el lenguaje imperativo tendríamos

$$
\begin{aligned}
\Gamma &= \text{conjunto de configuraciones} \\
\Gamma &= \langle \mathsf{expr} \rangle \\
\Gamma_t &= \text{expresiones sin redex, es decir en forma normal} \\
\Gamma_n &= \text{expresiones con redex} \\
\rightarrow &\subseteq \Gamma_n \times \Gamma
\end{aligned}
$$

y como se vió en alguno de los ejemplos de arriba $\rightarrow$ es no determinístico. Sin embargo, si $\rightarrow^*$ es la clausura reflexiva (y por renombre) y transitiva de $\rightarrow$, entonces podemos cerrar caminos que se separan.

Teorema 1. *Church-Rosser. Si $e \rightarrow^* e_0$ y $e \rightarrow^* e_1$, entonces existe e' tal que $e_0 \rightarrow^* e'$ y $e_1 \rightarrow^* e'$.*

También se la llama confluencia o propiedad del diamante:



Corolario 1. *Salvo renombre, toda expresión tiene a lo sumo una forma normal.*

Demostración. Supongamos que e_0 y e_1 son formas normales de e. Tenemos $e \rightarrow^* e_0$ y $e \rightarrow^* e_1$. Por teorema, existe e' tal que $e_0 \rightarrow^* e'$ y $e_1 \rightarrow^* e'$. Pero como e_0 y e_1 eran formas normales no tenían redices, por lo tanto e' sólo puede ser renombre de e_0 y de e_1, por lo tanto e_0 y e_1 son renombres entre sí. □

Pero no toda expresión tiene forma normal; veamos dos ejemplos patológicos:

1. Sea $\Delta = (\lambda x.xx)$, entonces
$$\Delta\Delta = (\lambda x.xx)\Delta \rightarrow \Delta\Delta \rightarrow \Delta\Delta \rightarrow \dots.$$
2. Sea $\Delta' = (\lambda x.xxy)$, entonces
$$\Delta'\Delta' = (\lambda x.xxy)\Delta' \rightarrow \Delta'\Delta' y \rightarrow \Delta'\Delta' yy \rightarrow \dots.$$

Y hay casos en las que la forma normal sólo se encuentra si se "ejecuta bien":

$$
\begin{aligned}
&(\lambda x.\lambda y.y)(\Delta\Delta) \rightarrow \lambda y.y \\
&(\lambda x.\lambda y.y)(\Delta\Delta) \rightarrow (\lambda x.\lambda y.y)(\Delta\Delta) \rightarrow \dots.
\end{aligned}
$$

1. Evaluacion

Hemos presentado la relación $\rightarrow$ que permite realizar cómputos, ejecuciones de un término lambda hasta llevarlo a su forma normal en caso de que la tenga. Pero ésa no es exactamente la idea de ejecución que implementan los lenguajes aplicativos.

La idea de ejecución (llamada evaluación) que se implementa habitualmente tiene las siguientes diferencias con la relación $\to$:

1. sólo se evalúan expresiones cerradas (es decir, sin variables libres)
2. es determinística
3. no busca formas normales sino formas canónicas

Existen varias definiciones de evaluación. Estudiaremos las dos más importantes: la evaluación (en orden) normal y la evaluación eager o estricta. La primera es la que ha dado lugar a los lenguajes de programación funcionales lazy (Haskell) y la segunda, a los estrictos (ML).

Como la evaluación busca una forma canónica de un término, las formas canónicas juegan el rol de ser "valores" de expresiones. La noción de forma canónica depende de la definición de evaluación. Se define una noción de forma canónica para la evaluación normal, y otra para la evaluación eager. En el caso del cálculo lambda coinciden: son las abstracciones (en otro lenguajes, veremos, dejan de coincidir).

Entonces tenemos: formas canónicas = abstracciones.

Como a partir de ahora nos concentraremos en expresiones cerradas, la siguiente propiedad ayuda para comparar las nociones de forma canónica y forma normal:

Propiedad 2. *Una aplicación cerrada no puede ser forma normal.*

Demostración. Sea e una aplicación cerrada. Sea $e = e_0 e_1 \dots$ en donde e_0 no es una aplicación. Como e es una aplicación, $n \geqslant 1$. Si e_0 fuera una variable, e no sería cerrada. Por lo tanto, es una abstracción y e contiene el redex $e_0 e_1$. Por lo tanto no es forma normal. □

Entonces, podemos concluir que una expresión cerrada que es forma normal es también forma canónica. El recíproco no vale, por ejemplo $\lambda x.(\lambda y.y)x$ es forma canónica cerrada pero no es forma normal. Peor aún, $\lambda x.\Delta\Delta$ es forma canónica cerrada y no tiene forma normal.

¿Por qué conformarse con una forma canónica en vez de continuar ejecutando hasta obtener una forma normal? Podemos entenderlo de la siguiente manera:

1. es razonable circunscribir la atención a expresiones cerradas (punto 1, apenas empezamos a hablar de evaluación) porque son las que parecen tener sentido,
2. una vez que se alcanzó una abstracción $\lambda v.M$, continuar evaluando implicaría evaluar M que puede no ser una expresión cerrada, puede contener a la variable v.

2. Evaluación en orden normal

La evaluación puede definirse utilizando semántica natural o big-step. En este tipo de semántica operacional, uno no describe un paso de ejecución, sino directamente una relación entre los términos y sus valores (que también son términos, son formas canónicas). Llamaremos $\Rightarrow$ a esta relación que está definida por las siguientes reglas:

$$\frac{}{\lambda v.e \Rightarrow \lambda v.e}\ (\text{abstracción}) \qquad \frac{e \Rightarrow \lambda v.e'' \qquad (e''/v \to e') \Rightarrow z}{ee' \Rightarrow z}\ (\text{aplicación})$$

Cuando afirmamos que $e \Rightarrow z$ se cumple, estamos diciendo que existe un árbol de derivación que demuestra $e \Rightarrow z$. Estos árboles usualmente son difíciles de graficar, por el tamaño de las expresiones intervinientes. Se suele usar indentación para expresar la estructura.

Por ejemplo, la evaluación normal de $(\lambda x.\lambda y.xx)\Delta$ se escribe:

$(\lambda x.\lambda y.xx)\Delta$	(es una aplicación, por lo tanto usamos la 2da regla, a continuación siguen los 2 hijos)
$\quad \lambda x.\lambda y.xx \Rightarrow \lambda x.\lambda y.xx$	(primer hijo, es una abstracción, por lo tanto usamos la 1er regla, no hay hijos)
$\quad \lambda y.\Delta\Delta \Rightarrow \lambda y.\Delta\Delta$	(segundo hijo, también abstracción, no hay hijos)
$\Rightarrow \lambda y.\Delta\Delta$	(terminó la prueba)

Otro ejemplo es la evaluación normal de $(\lambda x.x(\lambda y.xyy)x)(\lambda z.\lambda w.z)$:

$(\lambda x.x(\lambda y.xyy)x)(\lambda z.\lambda w.z)$	(aplicac. siguen 2 hijos: a,b)
$\quad \lambda x.x(\lambda y.xyy)x \Rightarrow \lambda x.x(\lambda y.xyy)x$	(a:abstracción, no hay hijos)
$\quad (\lambda z.\lambda w.z)(\lambda y.(\lambda z.\lambda w.z)yy)(\lambda z.\lambda w.z)$	(b:aplic. siguen 2 h: ba y bb)
$\qquad (\lambda z.\lambda w.z)(\lambda y.(\lambda z.\lambda w.z)yy)$	(ba: aplic. Siguen 2: baa,bab)
$\qquad\quad \lambda z.\lambda w.z \Rightarrow \lambda z.\lambda w.z$	(baa: abstracción,no hay h)
$\qquad\quad \lambda w.\lambda y.(\lambda z.\lambda w.z)yy \Rightarrow \lambda w.\lambda y.(\lambda z.\lambda w.z)yy$	(bab: abstracción,no hay h)
$\qquad \Rightarrow \lambda w.\lambda y.(\lambda z.\lambda w.z)yy$	(terminó ba)
$\qquad \lambda y.(\lambda z.\lambda w.z)yy \Rightarrow \lambda y.(\lambda z.\lambda w.z)yy$	(bb: abstracción,no hay hijos)
$\qquad \Rightarrow \lambda y.(\lambda z.\lambda w.z)yy$	(terminó b)
$\Rightarrow \lambda y.(\lambda z.\lambda w.z)yy$	(terminó la prueba)

La notación con indentación sola es difícil de leer, se mejora si se escriben corchetes "[.ª la izquierda encerrando los subárboles. En el caso de arriba habría un corchete que abarque las 10 líneas, otro que abarque sólo la línea 2, otro de la 3 a la 9, otro de la 4 a la 7, otro para la 8 sola, otro para la 5 sola y otro para la 6 sola. Los corchetes más grandes van más a la izquierda.

Hay expresiones, como $\Delta\Delta$ cuya evaluación no termina:

$(\lambda x.xx)(\lambda x.xx)$	(aplicación, siguen 2 hijos)
$\quad (\lambda x.xx) \Rightarrow (\lambda x.xx)$	(abstracción, no hay hijos)
$\quad (\lambda x.xx)(\lambda x.xx)$	(aplicación, siguen 2 hijos)
$\qquad (\lambda x.xx) \Rightarrow (\lambda x.xx)$	(abstracción, no hay hijos)
$\qquad (\lambda x.xx)(\lambda x.xx)$	(aplicación, siguen 2 hijos)
$\qquad\quad (\lambda x.xx) \Rightarrow (\lambda x.xx)$	(abstracción, no hay hijos)
$\qquad\quad (\lambda x.xx)(\lambda x.xx)$	(aplicación, siguen 2 hijos)
$\ldots$	

no podemos construir el árbol porque la evaluación no termina.

Podemos leer las reglas que definen $\Rightarrow$ como un algoritmo:

```
evalN :: Exp ->  Exp
evalN (Lam v e) = (Lam v e)
evalN (App e e') = case evalN e of
                        Lam v e'' ->  evalN (e''/v -> e')
```

que, por supuesto, requiere definir la sustitución para que sea un programa ejecutable. Los árboles mostrados más arriba pueden interpretarse como que la indentación corresponde a las llamadas recursivas. Es fácil comprobar que la aplicación de `evalN` a $\Delta\,\Delta$ no termina.

3. Evaluacion eager

La evaluación en orden normal a partir de $\Delta((\lambda x.x)(\lambda y.y))$ contrae dos veces el redex $(\lambda x.x)(\lambda y.y)$:

$(\lambda x.xx)((\lambda x.x)(\lambda y.y))$	(aplicación, siguen 2 hijos: a y b)
$\quad(\lambda x.xx) \Rightarrow (\lambda x.xx)$	(a: abstracción, no hay hijos)
$\quad((\lambda x.x)(\lambda y.y))((\lambda x.x)(\lambda y.y))$	(b: aplicación, siguen 2 h: ba y bb)
$\quad\quad(\lambda x.x)(\lambda y.y)$	(ba: aplicación, siguen 2 h: baa y bab)
$\quad\quad\quad(\lambda x.x) \Rightarrow (\lambda x.x)$	(baa: abstracción, no hay hijos)
$\quad\quad\quad(\lambda y.y) \Rightarrow (\lambda y.y)$	(bab: abstracción, no hay hijos)
$\quad\quad\Rightarrow (\lambda y.y)$	(terminó ba)
$\quad\quad(\lambda x.x)(\lambda y.y)$	(bb: aplicación, siguen 2 h: bba y bbb)
$\quad\quad\quad(\lambda x.x) \Rightarrow (\lambda x.x)$	(bba: abstracción, no hay hijos)
$\quad\quad\quad(\lambda y.y) \Rightarrow (\lambda y.y)$	(bbb: abstracción, no hay hijos)
$\quad\quad\Rightarrow (\lambda y.y)$	(terminó bb)
$\quad\Rightarrow (\lambda y.y)$	(terminó b)
$\Rightarrow (\lambda y.y)$	(terminó la prueba)

la repetición de la contracción se observa al ver que las líneas 4 a 7 y 8 a 11 son idénticas. Esto se podría evitar si se cambia el orden de evaluación, reduciendo el operando antes de sustituir en la regla de evaluación. La definición de evaluación eager, entonces, está dada por las siguientes reglas:

$$\frac{}{\lambda v.e \Rightarrow \lambda v.e}\ (\text{abstracción}) \qquad \frac{e \Rightarrow \lambda v.e'' \qquad e' \Rightarrow z' \qquad (e''/v \to z') \Rightarrow z}{ee' \Rightarrow z}\ (\text{aplicación})$$

Retomando la expresión anterior, con la evaluación eager tenemos:

$(\lambda x.xx)((\lambda x.x)(\lambda y.y))$	(aplicación, siguen 3 hijos: a, b y c)
$\quad(\lambda x.xx) \Rightarrow (\lambda x.xx)$	(a: abstracción, no hay hijos)
$\quad(\lambda x.x)(\lambda y.y)$	(b: aplicación, siguen 3 h: ba,bb y bc)
$\quad\quad(\lambda x.x) \Rightarrow (\lambda x.x)$	(ba: abstracción, no hay hijos)
$\quad\quad(\lambda y.y) \Rightarrow (\lambda y.y)$	(bb: abstracción, no hay hijos)
$\quad\quad(\lambda y.y) \Rightarrow (\lambda y.y)$	(bc: abstracción, no hay hijos)
$\quad\Rightarrow (\lambda y.y)$	(terminó b)
$\quad(\lambda y.y)(\lambda y.y)$	(c: aplicación, siguen 3 h: ca,cb y cc)
$\quad\quad(\lambda y.y) \Rightarrow (\lambda y.y)$	(ca: abstracción, no hay hijos)
$\quad\quad(\lambda y.y) \Rightarrow (\lambda y.y)$	(cb: abstracción, no hay hijos)
$\quad\quad(\lambda y.y) \Rightarrow (\lambda y.y)$	(cc: abstracción, no hay hijos)
$\quad\Rightarrow (\lambda y.y)$	(terminó c)
$\Rightarrow (\lambda y.y)$	(terminó la prueba)

También da lugar a un algoritmo de evaluación:

```
evalE :: Exp ->  Exp
evalE (Lam v e) = (Lam v e)
evalE (App e e') = case evalE e of
                     Lam v e'' ->  case evalE e' of
                        Lam w e_0 ->  evalE (e''/v -> Lam w e_0)
```

Así como hemos llamado de manera diferente a los programas (`evalN` y `evalE`) correspondientes a la evaluación normal e eager, escribiremos $\Rightarrow_N$ o $\Rightarrow_E$ para explicitar que nos referimos a la evaluación normal o eager.

Usualmente la evaluación eager es más rápida que la evaluación normal porque es común que en la normal se repitan contracciones que en la eager se hacen una sola vez. Puede ocurrir lo contrario: la eager puede evaluar un argumento que no

se utiliza. En ese caso, la eager es menos eficiente. Puede incluso ocurrir que la eager no termine en casos en los que la normal sí lo haga: $(\lambda x.\lambda y.y)(\Delta\Delta) \Rightarrow_N \lambda y.y$ mientras que en la evaluación eager, $(\lambda x.\lambda y.y)(\Delta\Delta)$ diverge.
Regla η. Un η-redex es una expresión de la forma $\lambda v.e\,v$ donde $v \notin FV(e)$. Gracias a la regla β, uno obtiene que $(\lambda v.e\,v)e'$ contrae a $e\,e'$ para toda expresión e'. Si uno asume que toda expresión lambda denota una función, $\lambda v.e\,v$ y e parecen comportarse extensionalmente igual: cuando se las aplica a e', ambas dan ee'. Esto motiva la regla η:

$$\frac{}{(\lambda v.ev) \to e} \text{ si } v \notin FV(e)$$

Esta regla no será tan estudiada como la β ya que no preserva significado en algunos lenguajes que veremos más adelante, en los que hay expresiones que no denotan funciones.

4. Semántica denotacional del cálculo lambda

Fue difícil. Por eso, es históricamente posterior a la operacional. Recordemos que cualquier expresión puede aplicarse a cualquier expresión. Por eso, si interpreto los términos en un conjunto C, y quiero que la aplicación del cálculo lambda se interprete como la aplicación usual en el metalenguaje, necesitaré que C sea igual o isomorfo a $C \to C$. En efecto, en $M\,M$ la primer M actúa como función de $C \to C$ y la segunda como elemento de C. Pero si tenemos un conjunto C que satisface eso, y sea $f \in C \to C$ cualquier función del dominio en sí mismo, podemos definir $p_f \in C \to$ C de la siguiente forma:

$$p_f\,x = f\,(x\,x)$$

Es más, ahora puedo aplicar p_f a sí misma: $p_f\,p_f = f\,(p_f\,p_f)$. Como se ve, partimos de una f cualquiera y le encontramos un punto fijo $(p_f\,p_f)$. O sea que el conjunto C debe ser isomorfo a $C \to C$ y debe satisfacer que toda función $f \in C \to C$ tenga punto fijo. Es difícil.

Hasta que Scott formuló los dominios, las funciones continuas y las soluciones a ecuaciones recursivas de dominios que utilicen $\to$, $\times$, $+$ y lifting.

Entonces se definió $D_\infty \cong [D_\infty \to D_\infty]$ es decir, D_∞ isomorfo a $[D_\infty \to D_\infty]$, donde esta flecha se refiere al espacio de funciones continuas , sabemos que todas ellas tienen punto fijo. No vamos a estudiar D_∞, pero sí utilizaremos que existe ese isomorfismo para definir la semántica denotacional. Sean

$$\phi \in D_\infty \to [D_\infty \to D_\infty]$$
$$\psi \in [D_\infty \to D_\infty] \to D_\infty$$

los isomorfismos tales que

$$\phi.\psi = id$$
$$\psi.\phi = id$$

Para las variables libres utilizaremos algo parecido a los estados, salvo que como acá no hay asignación, se lo denomina *ambiente*. Al conjunto de ambientes se denomina $Env = \langle \mathsf{var} \rangle \to D_\infty$; usaremos la letra griega η como metavariable para ambientes: $\eta \in Env$. La función semántica tendrá el tipo esperado: $[\![_]\!] \in \langle \mathsf{expr} \rangle \to Env \to D_\infty$.

Dado que no conocemos D_∞, sólo lo manipulamos a través de los isomorfismos ϕ y ψ. Veamos caso por caso la definición recursiva de la función semántica:

$$[\![v]\!]\eta = \eta\, v$$

En efecto, el valor de una variable está dado por el valor que tiene asociada en el ambiente.

Para la aplicación, escribiríamos

$$[\![e_0\, e_1]\!]\eta = ([\![e_0]\!]\eta)\,([\![e_1]\!]\eta)$$

respetando que la semántica de la aplicación sea la aplicación del metalenguaje. Pero acá no coinciden los tipos ya que $[\![e_0]\!]\eta \in D_\infty$, no es una función: tenemos que usar el isomorfismo para convertirlo en una función: $\phi([\![e_0]\!]\eta) \in [D_\infty \to D_\infty]$. La corregimos y queda así:

$$[\![e_0\, e_1]\!]\eta = \phi([\![e_0]\!]\eta)([\![e_1]\!]\eta)$$

De la misma forma, para la abstracción escribiríamos

$$[\![\lambda x.e]\!]\eta = \lambda d \in D_\infty \mapsto [\![e]\!][\eta|v : d]$$

respetando que la semántica de la abstracción sea la abstracción del metalenguaje. Nuevamente no dan los tipos ya que debería ser un D_∞, y es una función de $[D_\infty \to D_\infty]$. El otro isomorfismo nos rescata:

$$[\![\lambda x.e]\!]\eta = \psi(\lambda d \in D_\infty.[\![e]\!][\eta|v : d])$$

Por ejemplo, calculemos

$$\begin{aligned} [\![\lambda x.x]\!]\eta &= \psi(\lambda d \in D_\infty.[\![x]\!][\eta|x : d]) && \text{(ecuación de la abstracción)} \\ &= \psi(\lambda d \in D_\infty.d) && \text{(ecuación de la variable)} \end{aligned}$$

que es la identidad de D_∞, convertida por ψ en un elemento de D_∞. Si la aplicáramos a cualquier otro término:

$$\begin{aligned} [\![(\lambda x.x)\, M]\!]\eta &= \phi([\![\lambda x.x]\!]\eta)\,([\![M]\!]\eta) && \text{(ecuación de la aplicación)} \\ &= \phi(\psi(\lambda d \in D_\infty.d))\,([\![M]\!]\eta) && \text{(cálculo que ya hicimos para } \lambda x.x) \\ &= (\lambda d \in D_\infty.d)\,([\![M]\!]\eta) && \text{(los isomorfismos son inversas mutuas)} \\ &= [\![M]\!]\eta \end{aligned}$$

Dado que D_∞ sólo tiene funciones continuas, habría que demostrar que $[\![_]\!] \in \langle \mathsf{expr} \rangle \to Env \to D_\infty$ comprobando que el resultado $[\![e]\!]\eta$ son siempre continuas. El libro lo demuestra.

5. Teoremas

Teorema 2. *Coincidencia Si $\eta\, w = \eta'\, w$ para todo $w \in FV(e)$, entonces $[\![e]\!]\eta = [\![e]\!]\eta'$.*

Teorema 3. *Sustitución Si $[\![\delta\, w]\!]\eta = \eta' w$ para todo $w \in FV(e)$, entonces $[\![e/\delta]\!]\eta = [\![e]\!]\eta'$.*

Teorema 4. *Sustitución Finita $[\![e/v_1 \to e_1, \ldots, v_n \to e_n]\!]\eta = [\![e]\!][\eta|v_1 : [\![e_1]\!]\eta| \ldots |v_n : [\![e_n]\!]\eta]$.*

Teorema 5. *Renombre Si $v' \notin FV(e) - \{v\}$, entonces $[\![\lambda v'.(e/v \to v')]\!] = [\![\lambda v.e]\!]$.*

Todos estos teoremas se demuestran muy sencillamente por inducción en la estructura de la expresión. Se utilizan los isomorfismos, pero no se utiliza en absoluto que sean isomorfismos, es decir, no se utiliza que $\phi.\psi = id$ y $\psi.\phi = id$ en la demostración de estos teoremas.

En cambio, sí se las utiliza para los teoremas que siguen. Para demostrar la correctitud de la regla β, se usa $\phi.\psi = id$. Mientras que para demostrar la correctitud de la regla η, se usa $\psi.\phi = id$.

Propiedad 3. *(correctitud de la regla β): $[\![(\lambda v.e)\, e']\!] = [\![e/v \to e']\!]$.*

Demostración. dada en clase y en el libro. □

Propiedad 4. *(correctitud de la regla η): Si $v \notin FV(e)$, entonces $[\![\lambda v.ev]\!] = [\![e]\!]$.*

Demostración. dada en clase y en el libro. □

Sin conocer la construcción de D_∞, no podemos demostrar que $\Delta\,\Delta$ (donde $\Delta = (\lambda x.xx)$) tiene como semántica $\perp$. Asumiremos que $[\![\Delta\Delta]\!]\eta = \perp$.

¿A qué evaluación corresponde esta semántica? ¿A la normal o a la eager? A ninguna. Para verlo, calculemos $[\![\lambda y.\Delta\Delta]\!]\eta$:

$$\begin{aligned}[\![\lambda y.\Delta\Delta]\!]\eta &= \psi(\lambda d \in D_\infty.[\![\Delta\Delta]\!][\eta|y : d]) \\ &= \psi(\lambda d \in D_\infty.\perp)\end{aligned}$$

La función $\lambda d \in D_\infty.\perp$, es la que para cada elemento de D_∞ devuelve $\perp$. Recordemos que por definición de dominio de funciones, esta función es el $\perp$ de $[D_\infty \to D_\infty]$. O sea que nos queda $[\![\lambda y.\Delta\Delta]\!]\eta = \psi(\perp)$ Como ψ es un isomorfismo, tiene que mapear bottom en bottom, queda $[\![\lambda y.\Delta\Delta]\!]\eta = \perp$. Conclusión, la semántica denotacional de $\lambda y.\Delta\Delta$ es bottom. Eso demuestra que no coincide ni con la evaluación normal ni con la eager ya que ambas consideran a $\lambda y.\Delta\Delta$ como un valor, su evaluación no diverge.

6. Semántica denotacional normal

Hay que distinguir entre $\perp$ y $\lambda d \in D.\perp$. El primero corresponde a una expresión sin forma canónica, mientras que el segundo corresponderá a una (expresión que tenga) forma canónica como $\lambda y.\Delta\,\Delta$. Definimos V para interpretar a las expresiones que tienen forma canónica. Intuitivamente, V es el conjunto de valores, de la misma forma que a las formas canónicas las llamamos también valores en su momento. En cambio D, es el conjunto de resultados, que incluyen valores y otras cosas, en este caso, sólo se agrega bottom:

$$D = V_\perp, \text{ donde } V \cong [D \to D]$$

con los isomorfismos:

$$\begin{aligned}\phi &\in V \to [D \to D] \\ \psi &\in [D \to D] \to V\end{aligned}$$

Ahora, $\lambda d \in D.\perp$ es el bottom de $D \to D$, y por lo tanto $\psi(\lambda d \in D.\perp)$ es el bottom de V, pero no es el bottom de D.

Observar que:

$$\begin{aligned}\phi_{\perp\!\!\perp} &\in D \to [D \to D] \\ \iota_\perp.\psi &\in [D \to D] \to D\end{aligned}$$

Gracias a estas funciones podemos reescribir las tres ecuaciones que dimos al comienzo, reemplazando ϕ por $\phi_{\perp\!\!\perp}$ y ψ por $\iota_{\perp}.\psi$. La semántica de una expresión será un elemento de D, por lo tanto los entornos mapean variables a elementos de D: $Env = \langle \mathsf{var} \rangle \to D$.

$$\begin{aligned}
[\![_]\!] &\in \langle \mathsf{expr} \rangle \to Env \to D \\
[\![v]\!]\eta &= \eta v \\
[\![e_0\, e_1]\!]\eta &= \phi_{\perp\!\!\perp}([\![e_0]\!]\eta)([\![e_1]\!]\eta) \\
[\![\lambda x.e]\!]\eta &= (\iota_{\perp}.\psi)(\lambda d \in D.[\![e]\!][\eta|v : d])
\end{aligned}$$

Los teoremas que vimos antes siguen valiendo porque no dependen de las identidades. También vale la regla β, ya que la igualdad $\phi_{\perp\!\!\perp}.(\iota_{\perp}.\psi) = id$ sigue valiendo. Pero no vale la regla η, ya que la semántica de $\lambda y.\Delta\Delta y$ es $\psi(\lambda d \in D.\perp)$ mientras que la de $\Delta\Delta$ es $\perp$.

Es interesante observar que si bien la semántica denotacional no expresa un orden de evaluación ya que no es operacional, establece indirectamente el orden natural en que debe evaluarse. Por ejemplo, en el caso de la aplicación, si $[\![e_0]\!]\eta = \perp$, entonces $\phi_{\perp\!\!\perp}([\![e_0]\!]\eta) = \perp_{D\to D}$ por definición de $f_{\perp\!\!\perp}$. Ahora $\perp_{D\to D}$ es la función que siempre devuelve $\perp$, entonces $\phi_{\perp\!\!\perp}([\![e_0]\!]\eta)\,([\![e_1]\!]\eta) = \perp$, o sea, $[\![e_0\, e_1]\!]\eta = \perp$. Acabamos de comprobar que si $[\![e_0]\!]\eta = \perp$ entonces $[\![e_0\, e_1]\!]\eta = \perp$. Eso indica claramente que e_0 debe evaluarse para evaluar $e_0\, e_1$. ¿Por qué otra razón la no terminación de e_0 podría implicar siempre la no terminación de $e_0\, e_1$?

En cambio, por más que $[\![e_1]\!]\eta = \perp$, no necesariamente $[\![e_0\, e_1]\!]\eta = \perp$. Eso indica que e_1 no necesariamente debe evaluarse para evaluarse $e_0\, e_1$. Para ver un ejemplo en que $[\![e_1]\!]\eta = \perp$ y $[\![e_0\, e_1]\!]\eta \neq \perp$ lo dan $e_0 = \lambda y.\lambda x.x$ junto con $e_1 = \Delta\Delta$. Como mencionamos más arriba, la regla β vale, por lo tanto $[\![e_0\, e_1]\!]\eta = [\![\lambda x.x]\!]\eta \neq \perp$.

7. Semántica denotacional eager

La semántica denotacional eager, no debería satisfacer la regla β, ya que la evaluación eager no lo hace. En efecto, si la evaluación eager satisfaciera dicha regla, al evaluar $(\lambda y.\lambda x.x)\,(\Delta\Delta)$ debería dar lo mismo que evaluar $\lambda x.x$, pero ese no es el caso ya que este último término ya está en forma canónica, mientras que $(\lambda y.\lambda x.x)\,(\Delta\Delta)$ diverge al evaluarse eager. En la evaluación eager, nunca se alcanza a utilizar el operador $\lambda y.\lambda x.x$ ya que el operando $\Delta\,\Delta$, que debe evaluarse antes de sustituir y en $\lambda x.x$, diverge.

En general, un operador recién será utilizado cuando ya se haya evaluado el operando. El operador, entonces, no debe interpretarse como una función de $[D \to D]$, ya que sólo se aplica a formas canónicas. Debe interpretarse como una función de $V \to D$, ya que V es el dominio donde se interpretan las (expresiones que tienen) formas canónicas. Definimos

$$D = V_{\perp},\ \text{donde } V \cong [V \to D]$$

con el isomorfismo dado por:

$$\begin{aligned}
\phi &\in V \to [V \to D] \\
\psi &\in [V \to D] \to V
\end{aligned}$$

Observar que

$$\phi_{\perp\!\!\perp} \in D \to [V \to D]$$
$$\iota_{\perp}.\psi \in [V \to D] \to D$$

Gracias a estas funciones podemos reescribir las tres ecuaciones una vez más. El conjunto de entornos en este caso varía: $Env = \langle \mathsf{var} \rangle \to V$. Para comprender por qué, notemos que en la regla de la aplicación las variables son reemplazadas por formas canónicas.

$$
\begin{aligned}
[\![_]\!] &\in \langle \mathsf{expr} \rangle \to Env \to D \\
[\![v]\!]\eta &= \iota_{\perp}(\eta v) \\
[\![e_0\, e_1]\!]\eta &= (\phi_{\perp\!\!\perp}([\![e_0]\!]\eta))_{\perp\!\!\perp}\,([\![e_1]\!]\eta) \\
[\![\lambda x.e]\!]\eta &= (\iota_{\perp}.\psi)(\lambda z \in V.[\![e]\!][\eta|v : z])
\end{aligned}
$$

Los teoremas que vimos antes (salvo el de sustitución) siguen valiendo porque no dependen de las identidades. Pero no valen las reglas β ni η.

El teorema de sustitución puede adecuarse a ambientes que asocian valores de V a las variables, pero el significado del teorema resultante es más limitado. Nuevamente la semántica denotacional ahora puede verse que tanto si $[\![e_0]\!]\eta = \perp$ como o si $[\![e_1]\!]\eta = \perp$ tendremos que $[\![e_0\, e_1]\!]\eta = \perp$. Eso indica que tanto e_0 como e_1 deben evaluarse para evaluar $e_0\, e_1$.