# LENGUAJES APLICATIVOS

DANIEL FRIDLENDER Y HÉCTOR GRAMAGLIA

Extendemos el cálculo lambda:

$$
\begin{array}{rl}
\langle \mathsf{expr} \rangle ::= & \langle \mathsf{var} \rangle | \langle \mathsf{expr} \rangle \langle \mathsf{expr} \rangle | \lambda \langle \mathsf{var} \rangle . \langle \mathsf{expr} \rangle \\
& | \langle \mathsf{natconst} \rangle | \langle \mathsf{boolconst} \rangle \\
& | - \langle \mathsf{expr} \rangle | \langle \mathsf{expr} \rangle + \langle \mathsf{expr} \rangle | \ldots \text{ operadores aritméticos y relacionales} \\
& | \neg \langle \mathsf{expr} \rangle | \langle \mathsf{expr} \rangle \wedge \langle \mathsf{expr} \rangle | \ldots \text{ operadores lógicos} \\
& | \mathsf{if}\ \langle \mathsf{expr} \rangle\ \mathsf{then}\ \langle \mathsf{expr} \rangle\ \mathsf{else}\ \langle \mathsf{expr} \rangle \\
& | \mathsf{error} | \mathsf{typeerror} \\
\langle \mathsf{natconst} \rangle ::= & 0|1|2|\ldots \\
\langle \mathsf{boolconst} \rangle ::= & \mathsf{true}|\mathsf{false}
\end{array}
$$

## 1. Evaluación eager

Se definen las formas canónicas:

$$
\begin{array}{rl}
\langle \mathsf{cfm} \rangle ::= & \langle \mathsf{intcfm} \rangle | \langle \mathsf{boolcfm} \rangle | \langle \mathsf{funcfm} \rangle \\
\langle \mathsf{intcfm} \rangle ::= & \ldots | -2 | -1 | 0 | 1 | 2 | \ldots \\
\langle \mathsf{boolcfm} \rangle ::= & \langle \mathsf{boolconst} \rangle \\
\langle \mathsf{funcfm} \rangle ::= & \lambda \langle \mathsf{var} \rangle . \langle \mathsf{expr} \rangle
\end{array}
$$

Las expresiones que dan error serán consideradas como que divergen. Se puede diferenciar, pero requiere de otras (más y más complicadas) reglas de evaluación. En lo que sigue, $i$ corre sobre enteros, $b$ sobre booleanos, $z$ sobre formas canónicas.

Hay una regla para todas las formas canónicas, cada una de ellas evalúa a sí misma:

$$\frac{}{z \Rightarrow z}$$

Obviamente, esta regla incluye como caso particular el de la abstracción que se vió en el cálculo lambda puro, ya que las abstracciones son formas canónicas.

Para la aplicación, tenemos la regla de evaluación ya vista en el cálculo lambda:

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

A continuación las reglas que corresponden a las nuevas expresiones del lenguaje.

Si $e$ evalúa al (numeral correspondiente al) entero $i$, entonces $-e$ evalúa al (numeral correspondiente al entero opuesto). Lo mismo con el not:

$$\frac{e \Rightarrow [i]}{-e \Rightarrow [-i]} \qquad\qquad \frac{e \Rightarrow [b]}{\neg\, e \Rightarrow [\neg b]}$$

Como puede notarse, sólo definimos la evaluación para los casos correctos. No hay cómo derivar que $-\mathsf{true}$ evalúe a algo, ya que $\mathsf{true}$ no evalúa a un entero. A esto nos referíamos con la mención que hicimos más arriba de que no se diferencian los errores

Thiffany Bricet Fuentes de Abreu indicó algunos errores.

de los programas que no terminan. Simplemente, la relación $\Rightarrow$ no está definida en esos casos.Para operadores binarios, hay que tener cuidado con la división:

$$\frac{e \Rightarrow [i] \qquad e' \Rightarrow [i']}{e \oplus e' \Rightarrow [i \oplus i']} \oplus \in \{+, -, x, =, \neq, <, \geqslant, >, \leqslant\}$$

$$\frac{e \Rightarrow [i] \qquad e' \Rightarrow [i']}{e \oplus e' \Rightarrow [i \oplus i']} \; i' \neq 0, \oplus \in \{/, rem\}$$

Por primera vez, la división por 0 queda indefinida (en el mismo sentido que la no terminación, o $-\mathsf{true}$).

$$\frac{e \Rightarrow [b] \qquad e' \Rightarrow [b']}{e \owedge e' \Rightarrow [i \owedge i']}$$

Y por último, para el $\mathsf{if\ then\ else}$, tenemos dos reglas, una para cada una de las posibilidades (correctas) de la condición:

$$\frac{e \Rightarrow \mathsf{true} \qquad e' \Rightarrow z}{\mathsf{if}\, e \,\mathsf{then}\, e' \,\mathsf{else}\, e'' \Rightarrow z} \qquad \frac{e \Rightarrow \mathsf{false} \qquad e'' \Rightarrow z}{\mathsf{if}\, e \,\mathsf{then}\, e' \,\mathsf{else}\, e'' \Rightarrow z}$$

## 2. Evaluación normal

Para este lenguaje son las mismas formas canónicas que en el caso eager. Las reglas que hemos dado se repiten de manera exacta, salvo la de la aplicación que ya vimos en el cálculo lambda puro:

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

En realidad, vale la pena revisar las reglas para los operadores $\wedge$, $\vee$ y $\Rightarrow$, ya que en el contexto de un lenguaje con evaluación normal resultan poco adecuadas. Por ejemplo, la conjunción puede definirse:

$$\frac{e \Rightarrow \mathsf{false}}{e \wedge e' \Rightarrow \mathsf{false}}$$

que expresa que no hace falta evaluar $e'$ si $e$ evalúa a $\mathsf{false}$. Habría que completar la definición con otra regla para el caso en que $e$ evalúe a $\mathsf{true}$. De manera similar se puede proceder con $\vee$ y $\Rightarrow$. Esta definición de conectivas lógicas "lazy" también es aplicable a lenguajes eager.

Todas las conectivas lógicas pueden definirse con el $\mathsf{if\ then\ else}$. Por ejemplo, la conjunción en sus versiones "lazy" y "no lazy":

$$e \wedge e' = \mathsf{if}\, e \,\mathsf{then}\, e' \,\mathsf{else}\, \mathsf{false} \qquad e \wedge e' = \mathsf{if}\, e \,\mathsf{then}\, e' \,\mathsf{else}\, \mathsf{if}\, e' \,\mathsf{then}\, \mathsf{false}\, \mathsf{else}\, \mathsf{false}$$

## 3. Semántica denotacional eager

Recordemos que para el cálculo lambda puro teníamos

$$D = V_\perp \quad \text{donde } V \cong V \to D$$

Para ser más precisos que con la evaluación, se le agregan a D denotaciones para $\mathsf{error}$ y $\mathsf{typeerror}$, que llamaremos *error* y *typeerror* (¡pero no confundir lenguaje con metalenguaje!):

$$D = (V \oplus \{error, typeerror\})_\perp$$

Utilizaremos la siguiente notación:

$$\begin{aligned} \iota_{\mathsf{norm}} &= \iota_\perp \cdot \iota_0 && \in V \to D \\ err &= \iota_\perp(\iota_1\, error) && \in D \\ tyerr &= \iota_\perp(\iota_1\, typeerror) && \in D \end{aligned}$$

Para cualquier $f \in V \to D$, está la extensión $f_*$ de $f$ a todo $D$, $f_* \in D \to D$ definida por:

$$\begin{aligned} f_*\,(\iota_{\mathsf{norm}}\, z) &= f z \\ f_*\, err &= err \\ f_*\, tyerr &= tyerr \\ f_*\, \perp &= \perp \end{aligned}$$

Como se ve, $f_*$ propaga todo "mal comportamiento".

Recordemos ahora la definición de $V$ para el cálculo lambda puro (eager): $V \cong V \to D$ Ahora a $V$ hay que agregarle los valores correspondientes a las nuevas formas canónicas:

$$V \cong \mathbb{Z} + \mathbb{B} + (V \to D)$$

Se lo suele escribir así:

$$\begin{aligned} V &\cong V_{\mathsf{int}} + V_{\mathsf{bool}} + V_{\mathsf{fun}}, \quad \text{donde} \\ V_{\mathsf{int}} &= \mathbb{Z} \\ V_{\mathsf{bool}} &= \mathbb{B} \\ V_{\mathsf{fun}} &= V \to D \end{aligned}$$

donde queda quizá más claro que $V$ contiene valores provenientes de formas canónicas de enteros, booleanos y funciones.

Ahora el isomorfismo son $\phi$ y $\psi$ tales que

$$\phi \in V \to V_{\mathsf{int}} + V_{\mathsf{bool}} + V_{\mathsf{fun}} \qquad \psi \in V_{\mathsf{int}} + V_{\mathsf{bool}} + V_{\mathsf{fun}} \to V$$

También nos resultará conveniente utilizar la siguiente notación:

$$\begin{aligned} \iota_{\mathsf{int}} &= \psi \cdot \iota_0 \in V_{\mathsf{int}} \to V \\ \iota_{\mathsf{bool}} &= \psi \cdot \iota_1 \in V_{\mathsf{bool}} \to V \\ \iota_{\mathsf{fun}} &= \psi \cdot \iota_2 \in V_{\mathsf{fun}} \to V \end{aligned}$$

Para $\ell \in \{\mathsf{int}, \mathsf{bool}, \mathsf{fun}\}$, dada $f \in V_\ell \to D$, se denota por $f_\ell$ la extensión de $f$ a $V$:

$$\begin{aligned} f_\ell(\iota_\ell\, z) &= f z \\ f_\ell(\iota_{\ell'}\, z) &= tyerr, \text{ si } \ell \neq \ell' \end{aligned}$$

Estas funciones serán utilizadas para hacer el chequeo de tipos (dinámico). Observar que si tenemos una función $f \in V_\ell \to D$, podemos extenderla definiendo $f_\ell \in V \to D$ y luego volver a extenderla $(f_\ell)_* \in D \to D$.

Ecuaciones. Puesto que estamos en el lenguaje eager, los entornos mapean variables a valores: $Env = \langle \mathsf{var} \rangle \to V$. El tipo de la función semántica es el mismo que para el cálculo lambda $[\![\_]\!] \in \langle \mathsf{expr} \rangle \to Env \to D$, pero recordemos que ahora $D$ no es el mismo.

La semántica de las constantes es trivial (consideremos 0 o $\mathsf{true}$), salvo que hay que promover el resultado para que sea un $D$ (no sólo un $V_{\mathsf{int}}$ o $V_{\mathsf{bool}}$):

$$\begin{aligned} [\![0]\!]\eta &= \iota_{\mathsf{norm}}(\iota_{\mathsf{int}} 0) \\ [\![\mathsf{true}]\!]\eta &= \iota_{\mathsf{norm}}(\iota_{\mathsf{bool}} V) \end{aligned}$$

Para evaluar $-e$ se evalúa $e$ y se chequea que el resultado sea un entero (en caso contrario, el subíndice int se encargará de disparar un error de tipos) y que

no se haya producido ya algún error (en cuyo caso, el subíndice $*$ se encargará de propagarlo). Si todo anda bien se devuelve el entero correspondiente promoviendolo para que sea un $D$.

$$[\![-e]\!]\eta = (\lambda i \in V_{\mathsf{int}}.\iota_{\mathsf{norm}}(\iota_{\mathsf{int}} - i))_{\mathsf{int}*}([\![e]\!]\eta)$$

Lo mismo ocurre con la negación lógica:

$$[\![\neg e]\!]\eta = (\lambda b \in V_{\mathsf{bool}}.\iota_{\mathsf{norm}}(\iota_{\mathsf{bool}}\neg b))_{\mathsf{bool}*}([\![e]\!]\eta)$$

Y también con los operadores binarios:

$$[\![e_0 + e_1]\!]\eta = (\lambda i \in V_{\mathsf{int}}.(\lambda j \in V_{\mathsf{int}}.\iota_{\mathsf{norm}}(\iota_{\mathsf{int}} i + j))_{\mathsf{int}*}([\![e_1]\!]\eta))_{\mathsf{int}*}([\![e_0]\!]\eta)$$
$$[\![e_0 < e_1]\!]\eta = (\lambda i \in V_{\mathsf{int}}.(\lambda j \in V_{\mathsf{int}}.\iota_{\mathsf{norm}}(\iota_{\mathsf{bool}} i < j))_{\mathsf{int}*}([\![e_1]\!]\eta))_{\mathsf{int}*}([\![e_0]\!]\eta)$$

Más delicado es el caso de la división por cero que dispara un error:

$$[\![e_0/e_1]\!]\eta = (\lambda i \in V_{\mathsf{int}}.\left(\lambda j \in V_{\mathsf{int}}.\begin{cases} err & \text{si } j = 0 \\ \iota_{\mathsf{norm}}(\iota_{\mathsf{int}} i/j) & \text{c.c.} \end{cases}\right)_{\mathsf{int}*}([\![e_1]\!]\eta))_{\mathsf{int}*}([\![e_0]\!]\eta)$$

$$[\![\mathsf{if}\ e\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1]\!]\eta = \left(\lambda b \in V_{\mathsf{bool}}.\begin{cases} [\![e_0]\!]\eta & \text{si } b \\ [\![e_1]\!]\eta & \text{cc} \end{cases}\right)_{\mathsf{bool}*}([\![e]\!]\eta)$$

Los casos del cálculo lambda se adaptan como sigue:

$$\begin{aligned} [\![v]\!]\eta &= \iota_{\mathsf{norm}}(\eta v) \\ [\![e_0 e_1]\!]\eta &= (\lambda f \in V_{\mathsf{fun}}.(\lambda z \in V.fz)_*([\![e_1]\!]\eta))_{\mathsf{fun}*}([\![e_0]\!]\eta) \\ &= (\lambda f \in V_{\mathsf{fun}}.f_*([\![e_1]\!]\eta))_{\mathsf{fun}*}([\![e_0]\!]\eta) \\ [\![\lambda x.e]\!]\eta &= \iota_{\mathsf{norm}}(\iota_{\mathsf{fun}}(\lambda z \in V.[\![e]\!][\eta|v : z])) \end{aligned}$$

Finalmente, las ecuaciones triviales

$$\begin{aligned} [\![\mathsf{error}]\!]\eta &= err \\ [\![\mathsf{typeerror}]\!]\eta &= tyerr \end{aligned}$$

## 4. Semántica denotacional normal

La única diferencia con el eager está en la definición de $V_{\mathsf{fun}}$:

$$\begin{aligned} D &= (V + error, typeerror)_\perp \quad V \cong V_{\mathsf{int}} + V_{\mathsf{bool}} + V_{\mathsf{fun}}, \quad \text{donde} \\ V_{\mathsf{int}} &= \mathbb{Z} \\ V_{\mathsf{bool}} &= \mathbb{B} \\ V_{\mathsf{fun}} &= D \to D \end{aligned}$$

Ecuaciones. Los entornos ahora son mapeos de variables a $D$: $Env = \langle \mathsf{var} \rangle \to D$:

$$[\![]\!] \in \langle \mathsf{expr} \rangle \to Env \to D$$

Luego, las ecuaciones son todas iguales salvo las del cálculo lambda:

$$\begin{aligned} [\![v]\!]\eta &= \eta v \\ [\![e_0 e_1]\!]\eta &= (\lambda f \in V_{\mathsf{fun}}.f([\![e_1]\!]\eta))_{\mathsf{fun}*}([\![e_0]\!]\eta) \\ [\![\lambda x.e]\!]\eta &= \iota_{\mathsf{norm}}(\iota_{\mathsf{fun}}(\lambda d \in D.[\![e]\!][\eta|v : d])) \end{aligned}$$

Nuevamente, en el contexto del lenguaje normal, es adecuado definir las conectivas de manera lazy. Por ejemplo:

$$[\![e_0 \wedge e_1]\!]\eta = \left(\lambda b \in V_{\mathsf{bool}}.\begin{cases} [\![e_1]\!] & \text{si } b \\ \iota_{\mathsf{norm}}(\iota_{\mathsf{bool}}\mathsf{false}) & \text{c.c.} \end{cases}\right)_{\mathsf{bool}*}([\![e_0]\!]\eta)$$

Por otro lado, si se definen las conectivas lógicas en términos del if then else , no hace falta dar ecuaciones para cada una de ellas.

## 5. Extensiones

**5.1. Tuplas.** Se agregan expresiones para tuplas:

$$\begin{array}{rcl} \langle \mathsf{expr} \rangle & ::= & \langle\, \langle \mathsf{expr} \rangle, \ldots, \langle \mathsf{expr} \rangle \,\rangle \mid \langle \mathsf{expr} \rangle.\langle \mathsf{tag} \rangle \\ \langle \mathsf{tag} \rangle & ::= & \langle \mathsf{natconst} \rangle \end{array}$$

Evaluación Eager.

$$\begin{array}{rcl} \langle \mathsf{cfm} \rangle & ::= & \langle \mathsf{intcfm} \rangle \mid \langle \mathsf{boolcfm} \rangle \mid \langle \mathsf{funcfm} \rangle \mid \langle \mathsf{tuplecfm} \rangle \\ \langle \mathsf{tuplecfm} \rangle & ::= & \langle\, \langle \mathsf{cfm} \rangle, \ldots, \langle \mathsf{cfm} \rangle \,\rangle \end{array}$$

$$\frac{e_1 \Rightarrow z_1 \ldots e_n \Rightarrow z_n}{\langle\, e_1, \ldots, e_n \,\rangle \Rightarrow \langle\, z_1, \ldots, z_n \,\rangle}$$

$$\frac{e \Rightarrow \langle\, z_1, \ldots, z_n \,\rangle}{e.[k] \Rightarrow z_k}\ k < n$$

Evaluación Normal. Para la evaluación normal consideramos que una tupla está en forma canónica, sin importar si sus elementos lo están.

$$\langle \mathsf{tuplecfm} \rangle ::= \langle\, \langle \mathsf{expr} \rangle, \ldots, \langle \mathsf{expr} \rangle \,\rangle$$

La regla

$$\frac{}{\langle\, e_1, \ldots, e_n \,\rangle \Rightarrow \langle\, e_1, \ldots, e_n \,\rangle}$$

no se agrega ya que es un caso particular de la regla $z \Rightarrow z$ para formas canónicas.

$$\frac{e \Rightarrow \langle\, e_1, \ldots, e_n \,\rangle \qquad e_k \Rightarrow z}{e.[k] \Rightarrow z}\ k < n$$

Semántica Denotacional. Ahora tendremos $V \cong V_{\mathsf{int}} + V_{\mathsf{bool}} + V_{\mathsf{fun}} + V_{\mathsf{tuple}}$. Además se tiene $\iota_{\mathsf{tuple}} \in V_{\mathsf{tuple}} \to V$ y para cualquier función $f \in V_{\mathsf{tuple}} \to D$, $f_{\mathsf{tuple}} \in V \to F$. Semántica Denotacional Eager.

$$V_{\mathsf{tuple}} = V*$$

$$[\![\langle\, e_1, \ldots, e_n \,\rangle]\!]\eta = (\lambda z_1 \in V. \ldots (\lambda z_n \in V. \iota_{\mathsf{norm}}(\iota_{\mathsf{tuple}}\langle\, z_1, \ldots, z_n \,\rangle))_*([\![e_n]\!]\eta) \ldots)_*([\![e_1]\!]\eta)$$

$$[\![e.[k]]\!]\eta = \left(\lambda t \in V_{\mathsf{tuple}}. \begin{cases} \iota_{\mathsf{norm}}\, t.k & \text{si } k < \#t \\ tyerr & \text{c.c.} \end{cases}\right)_{\mathsf{tuple}*} ([\![e]\!]\eta)$$

Semántica Denotacional Normal.

$$V_{\mathsf{tuple}} = D*$$

$$[\![\langle\, e_1, \ldots, e_n \,\rangle]\!]\eta = \iota_{\mathsf{norm}}(\iota_{\mathsf{tuple}}\langle\, [\![e_1]\!]\eta, \ldots, [\![e_1]\!]\eta\rangle)$$

$$[\![e.[k]]\!]\eta = \left(\lambda t \in V_{\mathsf{tuple}}. \begin{cases} t.k & \text{si } k < \#t \\ tyerr & \text{c.c.} \end{cases}\right)_{\mathsf{tuple}*} ([\![e]\!]\eta)$$

**5.2. Definiciones locales y patrones.** Agregamos al lenguaje la posibilidad de definir localmente y notación para patrones:

$$\begin{array}{rcl} \langle \mathsf{expr} \rangle & ::= & \mathsf{let}\ \langle \mathsf{pat} \rangle \equiv \langle \mathsf{expr} \rangle, \ldots, \langle \mathsf{pat} \rangle \equiv \langle \mathsf{expr} \rangle\ \mathsf{in}\ \langle \mathsf{expr} \rangle | \lambda \langle \mathsf{pat} \rangle . \langle \mathsf{expr} \rangle \\ \langle \mathsf{pat} \rangle & ::= & \langle \mathsf{var} \rangle | \langle\, \langle \mathsf{pat} \rangle, \ldots, \langle \mathsf{pat} \rangle\, \rangle \end{array}$$

Así podemos escribir

$$\lambda \langle\, u, \langle\, v, w \rangle \rangle . uvw$$

en vez de

$$\lambda t.(t.0)(t.1.0)(t.1.1)$$

Para no tener que definir evaluación y semántica denotacional eager y normal para esta extensión, nos conformamos con ver que estas nuevas expresiones se pueden definir en términos de las que ya existían. Es sólo azúcar sintáctico:

$$\lambda \langle\, p_1, \ldots, p_n \,\rangle . e$$

es lo mismo que

$$\lambda v.\mathsf{let}\ p_1 \equiv v.0, \ldots, p_n \equiv v.[n-1]\ \mathsf{in}\ e$$

donde $v$ es una variable nueva (no ocurre libre en $e$ ni en ninguno de los patrones).

$$\mathsf{let}\ p_1 \equiv e_1, \ldots, p_n \equiv e_n\ \mathsf{in}\ e$$

es lo mismo que

$$(\lambda p_1 \ldots \lambda p_n . e) e_1 \ldots e_n$$

Aplicando repetidamente estas dos transformaciones podemos eliminar los patrones que no sean variables y las definiciones locales (let) obteniendo una expresión cuya semántica ya está definida.

Tener en cuenta que cuando $n = 0$, $\mathsf{let}\ p_1 \equiv e_1, \ldots, p_n \equiv e_n\ \mathsf{in}\ e$ quedaría $\mathsf{let}\ \mathsf{in}\ e$, esto en realidad es directamente la expresión $e$ (el $\mathsf{let}\ \mathsf{in}$ sin nada al medio es como si no existiera).

Si bien hemos definido $\mathsf{let}\ v \equiv e'\ \mathsf{in}\ e$ como azucar sintáctico, podríamos fácilmente deducir reglas de evaluación y ecuaciones semánticas; lo hagamos para que nos sirva de guía para las reglas y ecuaciones de las definiciones recursivas.

$$\frac{e' \Rightarrow z' \qquad e/v \to z' \Rightarrow_E z}{\mathsf{let}\ v \equiv e'\ \mathsf{in}\ e \Rightarrow_E z}$$

La ecuación semántica es igualmente sencila, si la semántica de $e'$ es un valor, entonces actualizamos el entorno para calcular la semántica de $e$:

$$[\![\mathsf{let}\ v \equiv e'\ \mathsf{in}\ e]\!]\eta = (\lambda z \in V.[\![e]\!][\eta \,|\, v : z])_*([\![e']\!]\eta)$$

Notemos que el entorno que usamos para calcular la semántica de $e'$ es el entorno original, es decir que las ocurrencias libres de $v$ en $e'$ reciben su semántica del entorno original y no del entorno modificado por la definición local; claramente esa es la diferencia entre una definición y una definición *recursiva*. En términos sintácticos, las ocurrencias libres de $v$ en $e'$ tambiñe son libres en $\mathsf{let}\ v \equiv e'\ \mathsf{in}\ e$.

**5.3. Recursión.** Recursión ha adoptado notaciones diferentes en las versiones eager y normal; mientras que en la eager se utiliza el letrec, en la normal se usa rec. Nos centremos por ahora en el lenguaje eager.

$$\langle \mathsf{expr} \rangle ::= \ldots \mid \mathsf{letrec}\ \langle \mathsf{var} \rangle \equiv \lambda \langle \mathsf{var} \rangle . \langle \mathsf{expr} \rangle, \ldots, \langle \mathsf{var} \rangle \equiv \lambda \langle \mathsf{var} \rangle . \langle \mathsf{expr} \rangle\ \mathsf{in}\ \langle \mathsf{expr} \rangle$$

El letrec permite hacer definiciones recursivas como

$$\mathsf{letrec}\, fact \equiv \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * fact(n-1)\ \mathsf{in}\ fact 10$$

A pesar de que la construcción sintáctica permite definir varias funciones mutuamente recursivas, nosotros daremos la regla de evaluación y la semántica asumiendo que definimos sólo una función recursiva.

Del ejemplo de factorial podemos deducir que en las definiciones recursivas $\mathsf{letrec}\, f \equiv \lambda v.e'\ \mathsf{in}\, e$, la variable $f$ de la función recursiva liga las ocurrencias libres de $f$ tanto en $e$ como en $e'$:

$$FV(\mathsf{letrec}\, f \equiv \lambda v.e'\ \mathsf{in}\, e) = (FV(\lambda\, v.e') \cup FV(e)) \setminus \{f\}$$

Esta es una diferencia con respecto a las definiciones no-recursivas y debemos tenerla en cuenta para entender la regla de evaluación.

Evaluación Eager. Podríamos pensar que la regla de evaluación para letrec es igual a la de let, pero al hacer el reemplazo de $f$ por $\lambda v.e'$ tendríamos, habitualmente, un término no-cerrado (por las llamadas recursivas de $f$ en $e'$). Es decir que al reemplazar $f$ por $\lambda v.e'$ debemos recordar la definición de $f$ por si se "activa" una llamada recursiva. Veamos ahora sí la regla:

$$\frac{e/(f \to \lambda v.\mathsf{letrec}\, f \equiv \lambda v.e'\ \mathsf{in}\, e') \Rightarrow z}{\mathsf{letrec}\, f \equiv \lambda v.e'\ \mathsf{in}\, e \Rightarrow z}\ f \neq v$$

Como la definición de $f$ tiene que ser una abstracción (esto es una restricción impuesta por la sintaxis) reemplazamos las ocurrencias de $f$ en $e$ por una abstracción, pero en vez de tener sólo el cuerpo, repetimos la definición recursiva de la misma de $f$ y desplegamos una vez la definición de $f$. Para entender cómo se efectúa la evaluación, supongamos que $e$ es $f\,\hat{e}$ siendo $\hat{z}$ el valor de $\hat{e}$, entonces deberemos evaluar $\mathsf{letrec}\, f \equiv \lambda v.e'\ \mathsf{in}\, (e'/v \to \hat{z})$.

Semántica Denotacional Eager. Al dar la semántica de let notamos que el entorno para el definiendo es el entorno original y eso implicaba que era una definición no-recursiva. Para definiciones recursivas querríamos:

$$[\![\mathsf{letrec}\, v \equiv \lambda u.e\ \mathsf{in}\, e']\!]\eta = [\![e']\!]\eta' \qquad \text{donde } \eta' = [\eta | v : [\![\lambda u.e]\!]\eta']$$

pero la definición de $\eta'$ está mal tipada, porque $[\![\lambda u.e]\!]\eta'$ pertenece a $D$ y por lo tanto no puede estar en el ambiente que es una función de $\langle \mathsf{var} \rangle$ en $V$. Esto es fácil de acomodar puesto que $\lambda u.e$ es un valor y por lo tanto su semántica es un elemento de $V$ inyectado apropiadamente; reescribimos

$$\eta' = [\eta | v : \iota_{\mathsf{fun}}(\lambda z \in V.[\![e]\!][\eta' | u : z])]$$

que está bien tipada. Sin embargo no podemos aplicar el teorema del menor punto fijo porque $Env$ no es un dominio. Una forma de solucionarlo sería definir una cadena de entornos con

$$\eta_0 = [\eta | v : \iota_{\mathsf{fun}}(\lambda z \in V.\bot_D)] \quad \text{y} \quad \eta_{i+1} = [\eta_i | v : \iota_{\mathsf{fun}}(\lambda z \in V.[\![e]\!][\eta_i | u : z])]$$

y tomar $\eta' = \sqcup_i \eta_i$. Otra opción es pensar que en esa cadena el único punto que va modificándose es $v$ y por lo tanto podemos definir un funcional $F \in [V_{fun} \to V_{fun}]$ donde el primer argumento (la función) es la semántica de $v$ en la iteración anterior:

$$F\ f\ z = [\![e]\!][\eta | v : \iota_{\mathsf{fun}} f | u : z]$$

Podemos probar que $F$ es continua y como $V_{fun}$ es un dominio el teorema de menor punto fijo es aplicable. Llegamos así a la definición oficial de la semántica de letrec:

$$[\![\mathsf{letrec}\ v \equiv \lambda u.e\ \mathsf{in}\ e']\!]\eta = [\![e']\!]\eta' \quad \text{donde}$$

$$\eta' = [\eta | v : \iota_{\mathsf{fun}} Y_{V_{\mathsf{fun}}}\ F] \quad \text{y} \quad F\ f\ z = [\![e]\!][\eta | v : \iota_{\mathsf{fun}} f | u : z]$$

En la definición de $\eta'$ utilizamos $Y_{V_{\mathsf{fun}}}$ para refererirnos al operador de menor punto fijo sobre $V_{\mathsf{fun}}$; es decir $Y\ F = \sqcup_i F^i \bot$ .

**Recursión en modalidad normal.** A diferencia de la modalidad eager, bajo la estrategia normal la recursión se incluye mediante un operador de punto fijo:

$$\langle \mathsf{expr} \rangle ::= \ldots \mid \mathsf{rec}\ \langle \mathsf{expr} \rangle$$

Contrastemos cómo escribiríamos (y aplicamos) el factorial con esta construcción:

$$\mathsf{rec}\ (\lambda f.\lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * f(n-1))\ 10$$

Podemos pensar que el argumento de rec debe ser la representación sintáctica del funcional asociado a la definición recursiva de factorial.

Evaluación Normal. Para entender la regla de evaluación pensemos que introdujimos rec como un operador de menor punto fijo; es decir esperamos que $\mathsf{rec}\, e$ tenga forma canónica si $e(\mathsf{rec}\, e)$ la tiene, y en ese caso debe ser la misma.

$$\frac{e(\mathsf{rec}\, e) \Rightarrow z}{\mathsf{rec}\, e \Rightarrow z}$$

Remarquemos que si $e$ tiene forma canónica que no es una abstracción, entonces $\mathsf{rec}\, e$ no tiene forma canónica.

Semántica Denotacional Normal. Al observar la regla de evalución de rec$e$ dijimos que $e$ debía evaluar a una abstracción; en la semántica denotacional debemos esperar que la semántica de $e$ sea una función (vista como un elemento de $D$) y en ese caso tomamos el menor punto fijo de dicha función:

$$[\![\mathsf{rec}\, e]\!]\eta = (\lambda f \in V_{\mathsf{fun}}.Y\, f)_{\mathsf{fun}*}([\![e]\!]\eta)$$