

CÁLCULO LAMBDA. TEOREMAS.

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

Teorema de Sustitución: Si $[[d w]]\eta = \eta' w$ para todo $w \in FV(e)$, entonces $[[e/d]]\eta = [[e]]\eta'$.

Teorema de Sustitución Finita: $[[e/v_1 \rightarrow e_1, \dots, v_n \rightarrow e_n]]\eta = [[e]]\eta|v_1: [[e_1]]\eta| \dots |v_n: [[e_n]]\eta$.

Teorema de Renombre: Si $v' \notin FV(e) - \{v\}$, entonces $[[\lambda v'.(e/v \rightarrow v')]] = [[\lambda v.e]]$.

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 $\phi \cdot \psi = \text{id}$ y $\psi \cdot \phi = \text{id}$ 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 $\phi \cdot \psi = \text{id}$. Mientras que para demostrar la correctitud de la regla η , se usa $\psi \cdot \phi = \text{id}$.

Regla η

Un η -redex es una expresión de la forma $\lambda v.ev$ donde $v \notin FV(e)$. Gracias a la regla β , uno obtiene que $(\lambda v.ev) e'$ contrae a $e e'$ para toda expresión e' . Si uno asume que toda expresión lambda denota una función, $\lambda v.ev$ y e parecen comportarse extensionalmente igual: cuando se las aplica a e' , ambas dan $e e'$. Esto motiva la

regla η :

----- si $v \notin FV(e)$
 $(\lambda v.e v) \rightarrow e$

Esta regla no va a ser 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.

Prop (correctitud de la regla β): $[[(\lambda v.e) e']] = [[e/v \rightarrow e']]$.

Dem: dada en clase y en el libro.

Prop (correctitud de la regla η): Si $v \notin FV(e)$, entonces $[[\lambda v.e v]] = [[e]]$.

Dem: dada en clase y en el libro.

LENGUAJES APLICATIVOS

Extendemos el cálculo lambda:

```
<exp> ::= <var> | <exp> <exp> |  $\lambda$ <var>.<exp>
        | <natconst> | <boolconst>
        | -<exp> | <exp> + <exp> | ... operadores aritméticos y relacionales
        |  $\neg$ <exp> | <exp>  $\wedge$  <exp> | ... operadores lógicos
        | if <exp> then <exp> else <exp>
        | error | typeerror
```

<natconst> ::= 0 | 1 | 2 | ...

<boolconst> ::= true | false

EVALUACIÓN EAGER

Se definen las formas canónicas:

<cfm> ::= <intcfm> | <boolcfm> | <funcfm>

```

<intcfm> ::= ... | -2 | -1 | 0 | 1 | 2 | ...
<boolcfm> ::= <boolconst>
<funcfm> ::= λ<var>.<exp>

```

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, *c/u* de ellas evalúa a sí misma:

```

-----
z => 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 forma canónica ya vista en el cálculo lambda:

```

e => λv.e"    e' => z'    (e"/v->z') => z
-----
e e' => 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*:

```

e => [i]          e => [b]
-----          -----
-e => [-i]        ¬e => [¬b]

```

Como puede notarse, sólo definimos la evaluación para los casos correctos. No hay cómo derivar que *-true* evalúe a algo, ya que *true* no es un entero. A esto nos referíamos con la mención que hicimos más arriba de que no se diferencian los errores de los programas que no terminan. Simplemente, la relación "*=>*" no está definida en esos casos.

Para operadores binarios, hay que tener cuidado con la división:

```

e => [i]    e' => [i']
-----    op ∈ {+, -, x, =, ≠, <, ≤, >, ≥} ∨ (i' ≠ 0 ∧ op ∈ {/, rem})
e op e' => [i op i']

```

Por primera vez, la división por 0 queda indefinida (en el mismo sentido que la no terminación, o *-true*).

```

e => [b]    e' => [b']
-----    op ∈ {∧, ∨, ⇒, ⇔}
e op e' => [b op b']

```

Y por último, para el *if then else*, tenemos dos reglas, una para cada una de las posibilidades (correctas) de la condición:

```

e => true    e' => z
-----
if e then e' else e" => z

e => false   e" => z
-----
if e then e' else e" => z

```

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:

```

e => λv.e"    (e"/v->e') => z

```

 $e \wedge e' \Rightarrow z$

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

$e \Rightarrow \text{false}$

 $e \wedge e' \Rightarrow \text{false}$

que expresa que no hace falta evaluar e' si e evalúa a false . Habría que completar la definición con otra regla para el caso en que e evalúe a true .

De manera similar se puede proceder con \vee y \Rightarrow . Esta definición de conectivas lógicas "lazy" también es aplicable a lenguajes normales.

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

$e \wedge e' = \text{if } e \text{ then } e' \text{ else false}$

$e \wedge e' = \text{if } e \text{ then } e' \text{ else if } e' \text{ then false else false}$

De la misma manera para \neg , \vee , \Rightarrow , \Leftrightarrow .

SEMÁNTICA DENOTACIONAL EAGER

 Recordemos que para el cálculo lambda puro teníamos

$D = V_{\perp}$ donde $V \approx V \rightarrow D$

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

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

Utilizaremos la siguiente notación:

$L_{\text{norm}} = L_{\text{bottom}} \ . \ L \in V \rightarrow D$

$\text{err} = L_{\text{bottom}} (L \ \text{error}) \in D$

$\text{tyerr} = L_{\text{bottom}} (L \ \text{typeerror}) \in D$

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

$f^* (L_{\text{norm}} \ z) = f \ z$

$f^* \ \text{err} = \text{err}$

$f^* \ \text{tyerr} = \text{tyerr}$

$f^* \ \perp = \perp$

Como se ve, f^* propaga todo "mal comportamiento".

Recordemos ahora la definición de V para el cálculo lambda puro (eager):

$V \approx V \rightarrow D$

Ahora a V hay que agregarle los valores correspondientes a las nuevas formas canónicas:

$V \approx Z + B + (V \rightarrow D)$

Se lo suele escribir así:

$V \approx V_{\text{int}} + V_{\text{bool}} + V_{\text{fun}}$

$V_{\text{int}} = Z$

$V_{\text{bool}} = B$

$V_{\text{fun}} = V \rightarrow D$

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

y funciones.

Ahora el isomorfismo son φ y ψ tales que

$$\begin{aligned}\varphi &\in V \rightarrow V_{\text{int}} + V_{\text{bool}} + V_{\text{fun}} \\ \psi &\in V_{\text{int}} + V_{\text{bool}} + V_{\text{fun}} \rightarrow V\end{aligned}$$

También nos resultará conveniente utilizar la siguiente notación:

$$\begin{aligned}L_{\text{int}} &= \psi \cdot L_0 \in V_{\text{int}} \rightarrow V \\ L_{\text{bool}} &= \psi \cdot L_1 \in V_{\text{bool}} \rightarrow V \\ L_{\text{fun}} &= \psi \cdot L_2 \in V_{\text{fun}} \rightarrow V\end{aligned}$$

Para $x \in \{\text{int}, \text{bool}, \text{fun}\}$, dada $f \in V_x \rightarrow D$, se denota por f_x la extensión de f a V :

$$\begin{aligned}f_x(L_x z) &= f z \\ f_x(L_y z) &= \text{tyerr}, \text{ si } y \neq x\end{aligned}$$

Estas funciones serán utilizadas para hacer el chequeo de tipos (dinámico).

Observar que si tenemos una función $f \in V_x \rightarrow D$, podemos extender definiendo $f_x \in V \rightarrow D$ y luego volver a extender $f_x^* \in D \rightarrow D$.

ECUACIONES

$$\begin{aligned}\text{Env} &= \langle \text{var} \rangle \rightarrow V \\ [[_]] &\in \langle \text{exp} \rangle \rightarrow \text{Env} \rightarrow D\end{aligned}$$

La semántica de θ o true , es trivial, salvo que hay que promover el resultado para que sea un D (no sólo un V_{int} o V_{bool}):

$$\begin{aligned}[[\theta]]_{\eta} &= L_{\text{norm}}(L_{\text{int}} \theta) \\ [[\text{true}]]_{\eta} &= L_{\text{norm}}(L_{\text{bool}} V)\end{aligned}$$

Para evaluar $-e$ se evalúa e y se chequea que dé 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 promoviendo para que sea un D .

$$[[-e]]_{\eta} = (\lambda i \in V_{\text{int}}. L_{\text{norm}}(L_{\text{int}} -i))_{\text{int}^*} ([[e]]_{\eta})$$

Lo mismo ocurre con la negación lógica:

$$[[\neg e]]_{\eta} = (\lambda b \in V_{\text{bool}}. L_{\text{norm}}(L_{\text{bool}} \neg b))_{\text{bool}^*} ([[e]]_{\eta})$$

Y también con los operadores binarios:

$$\begin{aligned}[[e_0 + e_1]]_{\eta} &= (\lambda i \in V_{\text{int}}. (\lambda j \in V_{\text{int}}. L_{\text{norm}}(L_{\text{int}} i + j))_{\text{int}^*} ([[e_1]]_{\eta}))_{\text{int}^*} ([[e_0]]_{\eta}) \\ [[e_0 < e_1]]_{\eta} &= (\lambda i \in V_{\text{int}}. (\lambda j \in V_{\text{int}}. L_{\text{norm}}(L_{\text{bool}} i < j))_{\text{int}^*} ([[e_1]]_{\eta}))_{\text{int}^*} ([[e_0]]_{\eta})\end{aligned}$$

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

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

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

Los casos del cálculo lambda se adaptan como sigue

$$\begin{aligned}[[v]]_{\eta} &= L_{\text{norm}}(\eta v) \\ [[e_0 e_1]]_{\eta} &= (\lambda f \in V_{\text{fun}}. (\lambda z \in V. f z)_{\text{fun}^*} ([[e_1]]_{\eta}))_{\text{fun}^*} ([[e_0]]_{\eta}) \\ &= (\lambda f \in V_{\text{fun}}. f_{\text{fun}^*} ([[e_1]]_{\eta}))_{\text{fun}^*} ([[e_0]]_{\eta})\end{aligned}$$

```
[[λx.e]]η = Lnorm (Lfun (λz∈V. [[e]][η|v:z]))
```

Finalmente, las ecuaciones triviales

```
[[error]]η = err
[[typeerror]]η = tyerr
```

SEMÁNTICA DENOTACIONAL NORMAL

La única diferencia con el eager está en la definición de Vfun:

```
D = (V + {error, typeerror})_⊥
```

```
V ≈ Vint + Vbool + Vfun
```

```
Vint = Z
```

```
Vbool = B
```

```
Vfun = D -> D
```

ECUACIONES

```
Env = <var> -> D
```

```
[[_]] ∈ <exp> -> Env -> D
```

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

```
[[v]]η = η v
[[e0 e1]]η = (λf∈Vfun.f ([[e1]]η))fun* ([[e0]]η)
[[λx.e]]η = Lnorm (Lfun (λd∈D. [[e]][η|v:d]))
```

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

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

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.

RECURSIÓN

Recursión ha adoptado notaciones diferentes en las versiones eager y normal: en la eager se utiliza el letrec y en la normal, rec:

```
<exp> ::= letrec <var> ≡ λ<var>.<exp>, ..., <var> ≡ λ<var>.<exp> in <exp>
      | rec <exp>
```

El letrec permite hacer definiciones recursivas como

```
letrec fact ≡ λn.if n = 0 then 1 else n * fact (n-1) in fact 10
```

mientras que rec se utiliza como el operador de punto fijo:

```
rec (λf.λn.if n = 0 then 1 else n * f (n-1)) 10
```

Evaluación Eager:

```
e'/(v->λu.letrec v ≡ λu.e in e) => z
----- v ≠ u,
      letrec v ≡ λu.e in e' => z
```

Evaluación Normal:

$e \text{ (rec } e) \Rightarrow z$

rec $e \Rightarrow z$

Semántica Denotacional Eager:

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

donde

$\eta' = [\eta | v: [[\lambda u. e]] \eta']$

pero la definición de η' está mal tipada, porque $[[\lambda u. e]] \eta'$ pertenece a D , no puede estar en el ambiente que es una función de $\langle \text{var} \rangle$ en V . Se reescribe

$\eta' = [\eta | v: \text{Lfun } (\lambda z \in V. [[e]] [\eta' | u: z])]$

que está bien tipada. Se logró resolver gracias a que el definiendo en el letrec es una abstracción. El problema ahora es que η' está definido en términos de sí mismo. Al no ser Env un dominio, no podemos aplicar el teorema del menor punto fijo.

Se reescribe

$\eta' = [\eta | v: \text{Lfun } f]$

$f = (\lambda z \in V. [[e]] [\eta | v: \text{Lfun } f | u: z])$

Ahora la f se definió en términos de sí mismo, pero se puede aplicar el teorema del menor punto fijo porque V_{fun} es un dominio.

Semántica Denotacional Normal:

$[[\text{rec } e]] \eta = (\lambda f \in V_{\text{fun}}. Y f) \text{fun}^* ([[e]] \eta)$

donde Y es el operador de menor punto fijo, $Y f = \sup f^i \perp$.

RECUSIÓN (acordarse de darlo después de tuplas y uniones, el año que viene)

Recusión ha adoptado notaciones diferentes en las versiones eager y normal: en la eager se utiliza el letrec y en la normal, rec:

$\langle \text{exp} \rangle ::= \text{letrec } \langle \text{var} \rangle \equiv \lambda \langle \text{var} \rangle. \langle \text{exp} \rangle, \dots, \langle \text{var} \rangle \equiv \lambda \langle \text{var} \rangle. \langle \text{exp} \rangle \text{ in } \langle \text{exp} \rangle$
 | rec $\langle \text{exp} \rangle$

El letrec permite hacer definiciones recursivas como

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

mientras que rec se utiliza como el operador de punto fijo:

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

Evaluación Eager:

$e' / (v \rightarrow \lambda u. \text{letrec } v \equiv \lambda u. e \text{ in } e) \Rightarrow z$

----- $v \neq u,$

letrec $v \equiv \lambda u. e \text{ in } e' \Rightarrow z$

Evaluación Normal:

$e \text{ (rec } e) \Rightarrow z$

rec $e \Rightarrow z$

Semántica Denotacional Eager:

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

donde

$\eta' = [\eta | v : [[\lambda u. e]] \eta']$

pero la definición de η' está mal tipada, porque $[[\lambda u. e]] \eta'$ pertenece a D , no puede estar en el ambiente que es una función de $\langle \text{var} \rangle$ en V . Se reescribe

$\eta' = [\eta | v : \text{Lfun } (\lambda z \in V. [[e]] [\eta' | u : z])]$

que está bien tipada. Se logró resolver gracias a que el definiendo en el letrec es una abstracción. El problema ahora es que η' está definido en términos de sí mismo. Al no ser Env un dominio, no podemos aplicar el teorema del menor punto fijo.

Se reescribe

$\eta' = [\eta | v : \text{Lfun } f]$
 $f = (\lambda z \in V. [[e]] [\eta | v : \text{Lfun } f | u : z])$

Ahora la f se definió en términos de sí mismo, pero se puede aplicar el teorema del menor punto fijo porque Vfun es un dominio.

Semántica Denotacional Normal:

$[[\text{rec } e]] \eta = (\lambda f \in \text{Vfun}. Y f) \text{fun}^* ([[e]] \eta)$

donde Y es el operador de menor punto fijo, $Y f = \sup f^i \perp$.