EVALUACIÓN EN ORDEN NORMAL (continuación)

¿Son equivalentes las dos definiciones de =>?

Prop: Para toda expresión cerrada b, b =N=> z sii b =IN=> z.
Dem: Veamos primero que si b =IN=> z entonces b =N=> z por inducción en la estructura de la derivación de b =IN=> z:
Si b =IN=> z, esto puede ocurrir gracias a la primer regla (en cuyo caso b = z = λv.e que es claramente la primer forma canónica en la reducción en orden normal a partir de sí misma, luego b =N=> z), o gracias a la segunda regla. En este último caso b = e e', y tenemos subderivaciones de e =IN=> λv.e" y de (e"/v->e') =IN=> z. Por hipótesis inductiva en ambas derivaciones, obtenemos e =N=> λv.e" y de (e"/v->e') =N=> z. Esto significa que las reducciones en orden normal a partir de e y de e"/v->e' tienen las siguientes formas:

1) e -> ... -> λv.e" -> ...
2) (e"/v->e') -> ... -> z -> ...

donde λv.e" y z son las primeras formas canónicas en la respectiva reducción en orden normal.
Como la reducción en orden normal elige siempre el redex que se encuentra más a la izquierda, la reducción 1) ignoraría un argumento que le demos a e:

1') b = e e' -> ... e' -> (λv.e") e'-> recién a partir de acá cambiaría!

¿Cómo continuaría la reducción en orden normal a partir de b? El siguiente paso sería contraer (λv.e") e' que es el redex más a la izquierda posible.

b = e e' -> ... e' -> (λv.e") e' -> (e"/v->e')

A partir de ahora continuaría con la reducción en orden normal de este último término, que es la reducción 2):

b = e e' -> ... e' -> (λv.e") e' -> (e"/v->e') -> ... -> z -> acá posiblemente continúe

Claramente z es la primer forma canónica en esta reducción, por lo tanto b =N=> z.

Veamos ahora que si b =N=> z entonces b =IN=> z.
Sea b -> ... -> z -> la reducción en orden normal a partir de b que tiene a z como primer forma canónica. Por inducción en la cantidad de pasos desde b hasta z. Si son 0 pasos es porque b = z es una abstracción. Gracias a la primer regla, obtenemos b =IN=> z.
Si son n > 0 pasos (y lo asumimos suerte para menos de n pasos) b es una aplicación (ya que es cerrada y no puede ser abstracción), b = e e'. Puede que por un rato la reducción en orden normal a partir de b sólo modifique el operador (eso ocurrirá mientras el operador sea a su vez una aplicación).

b = e e' -> e1 e' -> ... -> em e' -> ... -> z -> (donde el número de pasos de b a z es n)

Pero eso no puede ocurrir para siempre, dado que z no tiene esa forma. Tarde o temprano se debe llegar a em e' donde em es una abstracción (recordemos que mientras el operador sea una aplicación, al ser cerrado, no puede ser forma normal y por lo tanto tiene redex y puede continuar reduciéndose en la reducción en orden normal que estamos analizando). Sea entonces em el primero en la reducción de arriba que es una abstracción. Observamos que dicho tramo de la reducción deja e' intacto. Eso dice que la reducción en orden normal a partir de e copia dicho tramo:

e -> e1 -> ... -> em -> a partir de acá cambiaría!

donde em es la primer forma canónica en esta reducción. Como tiene menos de n pasos (es un subtramo propio de la reducción que va de b a z), por hipótesis inductiva e =IN=> em.

Hemos dicho además que em es una abstracción, sea em = λv.e". Reescribamos la reducción en orden normal a partir de b:

b = e e' -> e1 e' -> ... -> em e' = (λv.e") e' -> (e"/v->e') -> ... -> z ->

A partir de e"/v->e', tenemos la reducción en orden normal de e"/v->e', de la que z es la primer forma canónica. Este tramo también tiene menos de n pasos (ya que no incluye el paso (λv.e") e' -> (e"/v->e')), por hipótesis inductiva obtenemos (e"/v->e') =IN=> z.

Usando la segunda regla de =IN=>, obtenemos que e e' =IN=> z.

EVALUACION EAGER

La evaluación en orden normal a partir de Δ ((λx.x) (λy.y)) lleva 4 contracciones:

Δ ((λx.x) (λy.y)) -> (λx.x) (λy.y) ((λx.x) (λy.y)) -> (λy.y) ((λx.x) (λy.y)) -> (λx.x) (λy.y) -> (λy.y)

Pero la misma expresión puede reducirse más eficientemente:

Δ ((λx.x) (λy.y)) -> Δ (λy.y) -> (λy.y) (λy.y) -> (λy.y)

Esta estrategia que estamos utilizando, consistente en evaluar el argumento de un redex antes de contraerlo, se llama evaluación eager. Para definirla en términos de reducción, definimos una variante de la regla de contracción β:

regla βE-reducción:

-------------------- donde z es una forma canónica o una variable
(λv.e) z -> (e/v->z)

Esta regla contrae un βE-redex: es una expresión de la forma (λv.e) z donde z es una forma canónica o una variable.

Para una expresión cerrada e, decimos que e evalúa eager a z (e =E=> z) cuando hay una reducción de e a z en la que en cada paso se contrae el βE-redex que se encuentra más a la izquierda (y que no es subexpresión de una forma canónica).

Además de esta definición, se puede dar una definición inductiva:

Una regla para formas canónicas:

λv.e =IE=> λv.e

Una regla para la aplicación:

```
e =IE=> λv.e"    e' =IE=> z'    (e"/v->z') =IE=> z
-------------------------------------------------
                  e e' =IE=> z
```

Podemos, por ejemplo, computar el ejemplo dado al comenzar:

```
Δ ((λx.x) (λy.y))                    (aplicación, siguen 3 hijos: a, b y c)
  Δ => Δ                             (a: abstracción, no hay hijos)
  (λx.x) (λy.y)                      (b: aplicación, siguen 3 hijos: ba, bb y bc)
    (λx.x) => (λx.x)                 (ba: abstracción, no hay hijos)
    (λy.y) => (λy.y)                 (bb: abstracción, no hay hijos)
    (λy.y) => (λy.y)                 (bc: abstracción, no hay hijos)
  => (λy.y)                          (terminó b)
  (λy.y) (λy.y)                      (c: aplicación, siguen 3 hijos: ca, cb y cc)
    (λy.y) => (λy.y)                 (ca: abstracción, no hay hijos)
    (λy.y) => (λy.y)                 (cb: abstracción, no hay hijos)
    (λy.y) => (λy.y)                 (cc: abstracción, no hay hijos)
  => (λy.y)                          (terminó c)
=> (λy.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 e0 -> evalE (e"/v->Lam w e0)
```

Prop: Para toda expresión cerrada b, b =E=> z sii b =IE=> z.
Dem: muy similar a la hecha para la evaluación normal.

Usualmente 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:

(λx.λy.y) (Δ Δ) =N=> λy.y

mientras que en la evaluación eager, (λx.λy.y) (Δ Δ) diverge.

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 -> C. En efecto, en M M la primer M actúa como función de C -> C y la segunda como elemento de C. Pero si tenemos un conjunto C que satisface eso, y sea f ∈ C -> C cualquier función del dominio en sí mismo, podemos definir p ∈ C -> C de la siguiente forma:

p x = f (x x)

Es más, ahora puedo aplicar p a sí misma: p p = f (p p). Como se ve, partimos de una f cualquiera y le encontramos un punto fijo (p p). O sea que el conjunto C debe ser isomorfo a C -> C y debe satisfacer que toda función f ∈ C -> 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 ->, x, + y lifting.

Entonces se definió

Dinf ≈ Dinf -> Dinf

es decir, Dinf isomorfo a Dinf -> Dinf, donde esta flecha se refiere al espacio de funciones continuas (que alguna vez hemos escrito Dinf -c-> Dinf), sabemos que todas ellas tienen punto fijo.

No vamos a estudiar Dinf, pero vamos a utilizar que existe ese isomorfismo para definir la semántica denotacional.

Sean

φ ∈ Dinf -> (Dinf -> Dinf)
ψ ∈ (Dinf -> Dinf) -> Dinf

los isomorfismos tales que

φ . ψ = id
ψ . φ = id

Para las variables libres utilizaremos algo parecido a los estados, salvo que como acá no hay asignación, se lo denomina ambiente y usaremos la letra griega η (η ∈ <var> -> Dinf). El conjunto de ambientes se denomina Env.

[[_]] ∈ <exp> -> Env -> Dinf

Dado que no conocemos Dinf, sólo lo manipulamos a través de los isomorfismos φ y ψ:

[[v]]η = η v

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

[[e0 e1]]η = ([[e0]]η) ([[e1]]η)

respetando que la semántica de la aplicación sea la aplicación del metalenguaje. Pero acá no dan los tipos ya que ([[e0]]η) ∈ Dinf, no es una función, tenemos que usar el isomorfismo para convertirlo en una función: φ([[e0]]η) ∈ Dinf -> Dinf.

La corregimos y queda así:

[[e0 e1]]η = φ([[e0]]η) ([[e1]]η)

De la misma forma, para la abstracción escribiríamos

[[λx.e]]η = λd∈Dinf. [[e]][η|v:d]

respetando que la semántica de la abstracción sea la abstracción del metalenguaje. Nuevamente no dan los tipos ya que debería ser un Dinf, y es una función de Dinf -> Dinf. El otro isomorfismo nos rescata:

[[λx.e]]η = ψ(λd∈Dinf. [[e]][η|v:d])

Por ejemplo, calculemos

```
[[λx.x]]η = ψ(λd∈Dinf. [[x]][η|x:d])     (ecuación de la abstracción)
          = ψ(λd∈Dinf. d)                (ecuación de la variable)
```

que es la identidad de Dinf, convertida por ψ en un elemento de Dinf.

Si la aplicáramos a cualquier otro término:

```
[[(λx.x) M]]η = φ([[λx.x]]η) ([[M]]η)         (ecuación de la aplicación)
              = φ(ψ(λd∈Dinf. d)) ([[M]]η)    (cálculo que ya hicimos para λx.x)
              = (λd∈Dinf. d) ([[M]]η)         (los isomorfismos son inversas mutuas)
              = [[M]]η
```

Dado que Dinf sólo tiene funciones continuas, habría que demostrar que [[_]] ∈ <exp> -> Env -> Dinf comprobando que el resultado [[e]]η son siempre continuas. El libro (Reynolds) lo hace a través de la proposición 10.7.

Sin conocer la construcción de Dinf, no podemos demostrar que Δ Δ (donde Δ = (λx.xx)) tiene como semántica ⊥. Asumiremos que [[Δ Δ]]η = ⊥.

¿A qué evaluación corresponde esta semántica? ¿A la normal o a la eager?

A ninguna. Para verlo, calculemos [[λy.Δ Δ]]η:

```
[[λy.Δ Δ]]η = ψ(λd∈Dinf.[[Δ Δ]][η|y:d])
            = ψ(λd∈Dinf.⊥)
```

La función λd∈Dinf.⊥, es la que para cada elemento de Dinf devuelve ⊥. Recordemos que por definición de dominio de funciones, esta función es el ⊥' de Dinf -> Dinf. O sea que nos queda

[[λy.Δ Δ]]η = ψ(⊥')

Como ψ es un isomorfismo, tiene que mapear bottom en bottom, queda

[[λy.Δ Δ]]η = ⊥

Conclusión, la semántica denotacional de λy.Δ Δ es bottom. Eso demuestra que no coincide ni con la evaluación normal ni con la eager ya que ambas consideran a λy.Δ Δ como un valor, su evaluación no diverge.

SEMÁNTICA DENOTACIONAL NORMAL

Hay que distinguir entre ⊥ y λd∈D.⊥. El primero corresponde a una expresión sin forma canónica, mientras que el segundo corresponderá a una (expresión que tenga) forma canónica como λy.Δ Δ.

Definimos V para interpretar a las expresiones que tienen forma canónica. Intuitivamente, V = 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:

Definimos

D = V_⊥ donde V ≈ D -> D

con

φ ∈ V -> (D -> D)
ψ ∈ (D -> D) -> V

Ahora, λd∈D.⊥ es el bottom de D -> D, y por lo tanto ψ(λd∈D.⊥) es el bottom de V, pero no es el bottom de D.

Observar que:

φ_⊥⊥ ∈ D -> (D -> D)
Lbottom . ψ ∈ (D -> D) -> D

Gracias a estas funciones podemos reescribir las tres ecuaciones que dimos al comienzo, reemplazando φ por φ_⊥⊥ y ψ por Lbottom . ψ:

Env = <var> -> D
[[_]] ∈ <exp> -> Env -> D

[[v]]η = η v
[[e0 e1]]η = φ_⊥⊥([[e0]]η) ([[e1]]η)
[[λx.e]]η = (Lbottom . ψ)(λd∈D. [[e]][η|v:d])

Los teoremas que vimos antes siguen valiendo porque no dependen de las identidades. También vale la regla β, ya que la igualdad

φ_⊥⊥ . (Lbottom . ψ) = id

sigue valiendo. Pero no vale la regla η, ya que la semántica de λy.Δ Δ y es ψ(λd∈D.⊥) mientras que la de Δ Δ es ⊥.

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 [[e0]]η = ⊥, entonces φ_⊥⊥([[e0]]η) = ⊥_{D->D} por definición de f_⊥⊥. Ahora ⊥_{D->D} es la función que siempre devuelve ⊥, entonces φ_⊥⊥([[e0]]η) ([[e1]]η) = ⊥, o sea, [[e0 e1]]η = ⊥. Acabamos de comprobar que si [[e0]]η = ⊥ entonces [[e0 e1]]η = ⊥. Eso indica claramente que e0 debe evaluarse para evaluarse e0 e1. ¿Por qué otra razón la no terminación de e0 podría implicar siempre la no terminación de e0 e1?

En cambio, por más que [[e1]]η = ⊥, no necesariamente [[e0 e1]]η = ⊥. Eso indica que e1 no necesariamente debe evaluarse para evaluarse e0 e1.

Para ver un ejemplo en que [[e1]]η = ⊥ y [[e0 e1]]η ≠ ⊥ lo da e0 = λy.λx.x, e1 = Δ Δ. Como mencionamos más arriba, la regla β vale, por lo tanto [[e0 e1]]η = [[λx.x]]η ≠ ⊥.

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, la evaluar (λy.λx.x) (Δ Δ) debería dar lo mismo que evaluar λx.x, pero ese no es el caso ya que este último término ya está en forma canónica, mientras que (λy.λx.x) (Δ Δ) diverge al evaluarse eager. En la evaluación eager, nunca se alcanza a utilizar el operador λy.λx.x ya que el operando Δ Δ, que debe evaluarse antes de sustituir y en λx.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 -> D, ya que sólo se aplica a formas canónicas. Debe interpretarse como una función de V -> D, ya que V es el dominio donde se interpretan las (expresiones que tienen) formas canónicas.

Definimos

D = V_⊥ donde V ≈ V -> D

con

φ ∈ V -> (V -> D)
ψ ∈ (V -> D) -> V

Observar que:

φ_⊥⊥ ∈ D -> (V -> D)
Lbottom . ψ ∈ (V -> D) -> D

Gracias a estas funciones podemos reescribir las tres ecuaciones una vez más:

Env = <var> -> V
[[_]] ∈ <exp> -> Env -> D

[[v]]η = Lbottom (η v)
[[e0 e1]]η = (φ_⊥⊥([[e0]]η))_⊥⊥ ([[e1]]η)
[[λx.e]]η = (Lbottom . ψ)(λz∈V. [[e]][η|v:z])

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

Nuevamente la semántica denotacional ahora puede verse que tanto si [[e0]]η = ⊥ o si [[e1]]η = ⊥ tendremos que [[e0 e1]]η = ⊥. Eso indica que tanto e0 como e1 deben evaluarse para evaluar e0 e1.