LENGUAJES APLICATIVOS

Extendemos el cálculo lambda:

```
<exp> ::= <var> | <exp> <exp> | λ<var>.<exp>
      | <natconst> | <boolconst>
      | -<exp> | <exp> + <exp> | ... operadores aritméticos y relacionales
      | ¬<exp> | <exp> ∧ <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, =, ≠, <, ≤, >, ≥}
  e op e' => [i op i']


e => [i]    e' => [i']
---------------------- 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 e' => z
```

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

```
   e => false
----------------
e ∧ e' => 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 ∨ y ⇒. Esta definición de conectivas lógicas "lazy" también es aplicable a lenguajes eager.

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

```
e ∧ e' = if e then e' else false
e ∧ e' = if e then e' else if e' then false else false
```

De la misma manera para ¬, ∨, ⇒, ⇔.

SEMÁNTICA DENOTACIONAL EAGER

Recordemos que para el cálculo lambda puro teníamos

D = V_⊥ donde V ≈ V -> 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 + {error, typeerror})_⊥

Utilizaremos la siguiente notación:

Lnorm = Lbottom . L0 ∈ V -> D
err = Lbottom (L1 error) ∈ D
tyerr = Lbottom (L1 typeerror) ∈ D

Para cualquier f ∈ V -> D, está la extensíon f* de f a todo D, f* ∈ D -> D definida por:

f* (Lnorm z) = f z
f* err = err
f* tyerr = tyerr
f* ⊥ = ⊥

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

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

V ≈ V -> D

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

V ≈ Z + B + (V -> D)

Se lo suele escribir así:

V ≈ Vint + Vbool + Vfun
Vint = Z
Vbool = B
Vfun = V -> 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

φ ∈ V -> Vint + Vbool + Vfun
ψ ∈ Vint + Vbool + Vfun -> V

También nos resultará conveniente utilizar la siguiente notación:

Lint = ψ . L0 ∈ Vint -> V
Lbool = ψ . L1 ∈ Vbool -> V
Lfun = ψ . L2 ∈ Vfun -> V

Para x ∈ {int, bool, fun}, dada f ∈ Vx -> D, se denota por fx la extensión de f a V:

fx (Lx z) = f z
fx (Ly z) = tyerr, si y ≠ x

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

Observar que si tenemos una función f ∈ Vx -> D, podemos extender definiendo fx ∈ V -> D y luego volver a extender fx* ∈ D -> D.

ECUACIONES

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

La semántica de 0 o true, es trivial, salvo que hay que promover el resultado para que sea un D (no sólo un Vint o Vbool):

[[0]]η = Lnorm (Lint 0)
[[true]]η = Lnorm (Lbool V)

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 promoviendolo para que sea un D.

[[-e]]η = (λi∈Vint.Lnorm (Lint -i))int* ([[e]]η)

Lo mismo ocurre con la negación lógica:

[[¬e]]η = (λb∈Vbool.Lnorm (Lbool ¬b))bool* ([[e]]η)

Y también con los operadores binarios:

[[e0+e1]]η = (λi∈Vint.(λj∈Vint.Lnorm (Lint i+j))int* ([[e1]]η))int* ([[e0]]η)
[[e0<e1]]η = (λi∈Vint.(λj∈Vint.Lnorm (Lbool i<j))int* ([[e1]]η))int* ([[e0]]η)

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

$$[[e0/e1]]\eta = (\lambda i \in Vint.(\lambda j \in Vint. \begin{cases} err & \text{si } j=0 \\ Lnorm\ (Lint\ i/j) & \text{c.c.} \end{cases})int^*)int^*\ ([[e1]]\eta)\ int^*\ ([[e0]]\eta)$$

$$[[\text{if e then e0 else e1}]]\eta = (\lambda b \in Vbool. \begin{cases} [[e0]]\eta & \text{si b} \\ [[e0]]\eta & \text{c.c.} \end{cases})bool^*\ ([[e]]\eta)$$

Los casos del cálculo lambda se adaptan como sigue

[[v]]η = Lnorm (η v)
[[e0 e1]]η = (λf∈Vfun.(λz∈V.f z)* ([[e1]]η))fun* ([[e0]]η)
= (λf∈Vfun.f* ([[e1]]η))fun* ([[e0]]η)

[[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

$$[[e0 \wedge e1]]\eta = (\lambda b \in Vbool.\ \begin{cases} [[e1]]\eta & \text{si } b \\ Lnorm\ (Lbool\ false) & \text{c.c.} \end{cases})bool^*\ ([[e0]]\eta)$$

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.

TUPLAS

Se agregan expresiones para tuplas:

<exp> ::= <<exp>,...,<exp>> | <exp>.<tag>
<tag> ::= <natconst>

Evaluación

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

Evaluación Eager

<tuplecfm> ::= <<cfm>,...,<cfm>>

 e1 => z1 ... en => zn

<e1,..,en> => <z1,..,zn>

e => <z1,..,zn>
--- k < n, [k] es la notación para k en el lenguaje
  e.[k] => zk

Evaluación Normal

<tuplecfm> ::= <<exp>,...,<exp>>

La regla

```
------------------------
<e1,..,en> => <e1,..,en>
```

no se agrega ya que es un caso particular de la regla z => z para formas canónicas.

```
e => <e1,..,en>     ek => z
--------------------------- k < n, [k] es la notación para k en el lenguaje
        e.[k] => z
```

Semántica Denotacional

Ahora tendremos

V ≈ Vint + Vbool + Vfun + Vtuple

Además se tiene Ltuple ∈ Vtuple -> V y para cualquier función f ∈ Vtuple -> D, ftuple ∈ V -> F.

Semántica Denotacional Eager

Vtuple = V*

[[<e1,..,en>]]η =

(λz1∈V. .. (λzn∈V. Lnorm (Ltuple <z1,..,zn>))* ([[en]]η) ..)* ([[e1]]η)

$$[[e.[k]]]\eta = (\lambda t \in Vtuple.\ \begin{cases} Lnorm\ tk & si\ k < \#t \\ tyerr & c.c. \end{cases})tuple^*\ ([[e]]\eta)$$

Semántica Denotacional Normal

Vtuple = D*

[[<e1,..,en>]]η = Lnorm (Ltuple <[[e1]]η,..,[[e1]]η>)

$$[[e.[k]]]\eta = (\lambda t \in Vtuple.\ \begin{cases} tk & si\ k < \#t \\ tyerr & c.c. \end{cases})tuple^*\ ([[e]]\eta)$$

DEFINICIONES LOCALES Y PATRONES

Agregamos al lenguaje la posibilidad de definir localmente y notación para patrones:

```
<exp> ::= let <pat> ≡ <exp>, ..., <pat> ≡ <exp> in <exp>
        | λ<pat>.<exp>
```

<pat> ::= <var> | <<pat>,..,<pat>>

Así podemos escribir λ<u,<v,w>>.u v w en vez de λt. 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:

λ<p1,..,pn>.e es lo mismo que λv.let p1 ≡ v.0, .., pn ≡ v.[n-1] in e

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

let p1 ≡ e1, .., pn ≡ en in e es lo mismo que (λp1..λpn.e) e1 .. en

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, let p1 ≡ e1, .., pn ≡ en in e quedaría let in e, esto en realidad es directamente la expresión e (el let in sin nada al medio es como si no existiera).

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 (rec e) => z
----
  rec e => z

Semántica Denotacional Eager:

[[letrec v ≡ λu.e in e']]η = [[e']]η'

donde

η' = [η|v:[[λu.e]]η']

pero la definición de η' está mal tipada, porque [[λu.e]]η' pertenece a D, no

puede estar en el ambiente que es una función de <var> en V. Se reescribe

η' = [η|v:Lfun (λz∈V.[[e]][η'|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

η' = [η|v:Lfun f]
f = (λz∈V.[[e]][η|v: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:

[[rec e]]η = (λf∈Vfun.Y f)fun* ([[e]]η)

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