REFERENCIAS Y ESTADOS EN UN LENGUAJE APLICATIVO EAGER

Una forma de combinar los paradigmas aplicativo e imperativo es introducir en un lenguaje funcional eager las referancias (o locaciones, que apuntan a un lugar de memoria) como un tipo particular de valor. A este lenguaje le llamaremos Iswim, y fue introducido por Peter Landin.

Las referencias serán para nosotros una abstracción del concepto de dirección de memoria, relativo al hardware. Utilizaremos el intervalo natural [0,n-1] para representar los lugares de memoria ocupados en un determinado momento de la ejecución, lo que constituirá el dominio de la función "estado", que devolverá un valor cada vez que se le pasa una referencia de su dominio. Si R representa el domino del estado s, entonces newref R será la primer dirección de memoria que no ocupa R, o sea newref [0,n-1]= n. Mediante Σ denotaremos a la familia de estados, o sea

Rf = N

Σ = U { [0,n-1] -> V : n = 0,1,...}

El lenguaje Iswim extiende al lenguaje aplicativo eager mediante las siguientes construcciones:

<exp> ::= skip | ref <exp> | val <exp> | <exp> := <exp> | <exp> =ref <exp>

EVALUACIÓN

Sumamos como formas canónicas a las referencias, y a una nueva forma canónica, destinada a representar el cómputo de un programa que no devueleve ningún valor, sino que sólo mofifica el estado (como por ejemplo skip o v:=e):

<cnf> ::= Rf | <>

El predicado de evaluación tendrá la siguiente forma, que refleja no sólo el cómputo de un valor sino además la modificación del estado:

s,e => z,s'

Todas las reglas aplicativas del lenguaje eager se incorporan con la indicación explícita de cómo se transforma el estado. Por ejemplo la regla

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

se transforma en:

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

Las reglas de las expresiones que modifican o leen el estado son:

```
--------------
s,skip => <>,s

  s,e => z, s'
-------------------- (r = newref(dom s))
s,ref e => r,[s|r:z]

  s,e => r, s'
-----------------
```

```
s,val e => s r,s'

(Aunque (s r) no es exactamente una forma canónica!)

s,e => r, s'  s',e' => z',s''
----------------------------
s,e:=e' => <>,[s|r:z']

s,e => r, s'  s',e' => r',s''
----------------------------
s,e =ref e' => [r=r'],s''


La evaluación de una expresión comienza con el estado [], es decir el estado con dominio vacío.



FRASES TÍPICAS DE LOS LENGUAJES IMPERATIVOS
-------------------------------------------

Típicas construcciones imperativas son tartadas ahora como syntactic sugar:

newvar v := e in e'  =def  let v = ref e in e'

e;e' =def  let v = e in e'   (donde v no es libre en e')

while e do e'  =def  letrec w = λv.if e then e'; w skip else skip in w skip  (donde v no es libre en e ni
en e')


PROPIEDADES
-----------

1. Si

s,e => z,s'
s',e' => z',s''

entonces   s,e;e' => z',s''

2. Si

s,e => z,s'
[s'|v:z],(e'/v->r) => z',s''

entonces    s,newvar v:=e in e' => z',s''

3. Si

s,e => false,s'

entonces s,while e do e' => <>,s'

4. Si

s,e => true,s'
s',e';while e do e' => z',s''

entonces s,while e do e' => z',s''


Prueba de 3 y 4
---------------

        while e do e'  =def letrec w = λv.c in w skip

donde v no es libre en e ni en e' y
```

```
        c =def if e then e'; w skip else skip

Sea c* = letrec w = λv.c in c. Entnces para evaluar la frase original debemos evaluar

        (w skip)/(w -> λv.c*)  =   (λv.c*) skip

Luego tenemos que evaular while e do e' es lo mismo que evaluar (λv.c*) skip. (Propiedad (1))

Continuando, necesitamos saber evaluar (c*/v-><>), ya que:

s,(λv.c*) => (λv.c*),s   s,skip => <>,s  s,(c*/v-><>) => ?,?
_____________________________________________________________

                s',(λv.c*) skip  =>  ?,?

Veamos que:

a) s',(c*/v-><>) => <>,s' (lo que necesitamos para 3) está dado por la hipótesi s,e => false,s',
b) s',(c*/v-><>) => z',s'' (lo que necesitamos para 4) está dado por las hipótesis

s,e => true,s'
s',e';while e do e' => z',s''.

En efecto, (c*/v-><>) = letrec w = λv.c in c y para evaluarlo debemos evaluar

c/w->λv.c*  =  if e then e';(λv.c*) skip else skip
            =  if e then e';while e do e' else skip (por propiedad (1))

Luego, por las regla de evaluación del if sale a) y b).
```