Auxiliar Extra C3
CC41A - CC54A
Lenguajes de Programación

Profesor: Éric Tanter
Aux: Richard Ibarra, Oscar E. A. Callaú
{*etanter,ribarra,oalvarez*}*@dcc.uchile.cl*

`Santiago - Chile, Jun/25/2008`

1. Recursión

1.1. letrec

Se le ha pedido que agregue la siguiente funcionlidad a su interprete, el `letrec` con sintaxis:

```
expr ::= ..
      |  {letrec {{var expr}*} body}
```

con la siguiente semántica:

```
{letrec {{var1 expr1}
         ..
         {varN exprN}}
 body-expr
}
```

- todas las `var1,..,varN` estan ligadas en el ambiente de cada `exprK`
- todas las `var1,..,varN` estan ligadas en el ambiente de `body-expr`

Pero usted descubre que existen al menos dos casos de borde con el uso del `letrec`, cuales son estas? Justifique su repuesta y proponga una solución.

1.2. cyclic environment

Benancio, un alumno despistado del curso, se le acerca a pedirle que le explique porque en la siguiente implementación:

```
(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (numV 1729)]
            . . .)
       ))
```

se usa un valor 1729? Que significa ese valor? es una fecha es un mensaje críptico?

Respuesta: ...

Benancio (de nuevo) insiste en que ese valor tiene un significado especial y que el resultado de la interpretación de una expresión recursiva puede retornar ese valor antes de encontrar su valor correcto. Como refuta la aseveración de Benancio?

2. Opciones de Representación

2.1. Ambiente

Hasta ahora hemos visto la representación del ambiente como una estructura de mapeo entre variables asociadas a sus valores, explique en se que basa el libro *PLAI* para justificar la construcción de un ambiente como función?

2.2. Representando listas

Implemente la representación de listas como funciones. Ej.

```
((list- 1 (list- 2 (empty-))) 'head) --> 1
(((list- 1 (list- 2 (empty-))) 'tail) 'head) --> 2
```

Suponga que nunca se haran llamadas inapropiadas a la representación *empty* de una lista.

3. Pauta, Recusión

3.1. letrec

El problema se presenta en los siguientes casos:

```
{letrec
{{f f}}
f
}

{letrec
{{f g}
{g f}}
g
}
```

Lo que sucede en estos casos es que cuando usamos un letrec, indicamos al interprete que esta comenzando una definición de una expresión recursiva, que en los casos mencionados genera un ciclo infinito. Ej. Cuando evalua el `body` del primer `letrec`: f, va ha buscarlo al ambiente y se da cuenta que depende de f y entonces va ha buscarlo de nuevo al ambiente (ya que tengo un ambitente ciclico) y este nuevo f (en si es el mismo) depende de f, que a veces depende de . . . , En el segundo caso tenemos el problema con una recursión indirecta, donde vemos que g depende de f y f depende de g y asi sucesivamente . . .

Una solución propuesta es restringuir la escritura de cada `exprK` del `letrec` sintacticamente a ser un *procedure*. Esto hace que cada vez evalua una `exprK` tiene que construir una clausura y la clausura no puede ser aplicada hasta que interpretemos por completo el cuerpo del `letrec`, que es cuando el ambiente va ha estar en un estado estable.

3.2. cyclic environment

La respuesta a nuestro amigo Benancio es que, en la creación de un ambiente cíclico necesitamos primero extender el ambiente con la ligación a un nuevo identificador, este identificador va ser el resultado de la interpretación y como por ahora no conocemos su resultado, le asignamos un valor por *defecto* (el 1729) que luego va ser reemplazado por su valor correcto.

Simplemente la interpretación no puede tomar el valor de 1729 como resultado antes de obtener su valor real, porque hemos restringido sintacticamente a que las expresiones recursivas sean funciones.

4. Opciones de Representación

4.1. Ambiente

Uso de funciones parciales (se deja la justificación como propuesto)

4.2. Representando listas

```
(define list-
  (lambda (x xs)
    (lambda (f)
      (cond
        ((eq? 'head f) x)
        ((eq? 'tail f) xs)
        (else error 'no_fx)))))
(define (empty-)
  error 'empty-list
  )

((list- 1 (list- 2 (empty-))) 'head)
(((list- 1 (list- 2 (empty-))) 'tail) 'head)
```