Facultad de Ciencias, UNAM
Lenguajes de Programación
Examen Parcial 2

Rubí Rojas Tania Michelle

07 de enero de 2021

1. Evalúa la siguiente expresión usando representación de ambientes en todos los casos. Es necesario expresar el ambiente final (stack) para evaluación glotona y perezosa; además de la expresión completa a evaluar en cada uno de los incisos anteriores, antes de dar el resultado final de tales evaluaciones.

```
{with {x {+ 2 2}}
  {with {y {+ 1 2}}
    {with {z 3}}
      {with {foo {fun {x} {+ x {+ y z}}}}
        {with {x 3}
          {with {y {+ 2 2}}
            {with {z {+ 1 1}}
              {with {mas-foo {fun {y} {* 1 {* x y}}}}
                {with {x 2}
                  {mas-foo 3}}}}}}}}}
```

- Evaluación perezosa y alcance estático.
  Solución: La expresión que debemos evaluar es `{mas-foo 3}`, por lo que

$$
\begin{aligned}
\texttt{\{mas-foo 3\}} &= \texttt{\{\{fun \{y\} \{* 1 \{* x y\}\}\} 3\}} \\
&= \texttt{\{* 1 \{* x 3\}\}} \\
&= \texttt{\{* 1 \{* 3 3\}\}} \\
&= \texttt{\{* 1 9\}} \\
&= 9
\end{aligned}
$$

x	2
mas-foo	[closureV: y, {* 1 {* x y}}, env-ant: (y 3), env7]
z	{+ 1 1}
y	{+ 2 2}
x	3
foo	[closureV: x,{+ x {+ y z}}, env-ant:env3]
z	3
y	{+ 1 2}
x	{+ 2 2}

Tabla 1: Ambiente final

- Evaluación perezosa y alcance dinámico.
  Solución: La expresión que debemos evaluar es `{mas-foo 3}`, por lo que

`{mas-foo 3}` = `{{fun {y} {* 1 {* x y}}} 3}`
= `{* 1 {* x 3}}`
= `{* 1 {* 2 3}}`
= `{* 1 6}`
= 6

x	2
mas-foo	{fun {y} {* 1 {* x y}}}
z	{+ 1 1}
y	{+ 2 2}
x	3
foo	{fun {x} {+ x {+ y z}}}
z	3
y	{+ 1 2}
x	{+ 2 2}

Tabla 2: Ambiente final

- Evaluación glotona y alcance estático.
  Solución: La expresión que debemos evaluar es `{mas-foo 3}`, por lo que

`{mas-foo 3}` = `{{fun {y} {* 1 {* x y}}} 3}`
= `{* 1 {* x 3}}`
= `{* 1 {* 3 3}}`
= `{* 1 9}`
= 9

x	2
mas-foo	[closureV: y, {* 1 {* x y}}, env-ant: ((y 3), env7)]
z	2
y	4
x	3
foo	[closureV: x,{+ x {+ y z}}, env-ant: env3]
z	3
y	3
x	4

Tabla 3: Ambiente final

- Evaluación glotona y alcance dinámico.
  Solución: La expresión que debemos evaluar es `{mas-foo 3}`, por lo que

`{mas-foo 3}` = `{{fun {y} {* 1 {* x y}}} 3}`
= `{* 1 {* x 3}}`
= `{* 1 {* 2 3}}`
= `{* 1 6}`
= 6

x	2
mas-foo	{fun {y} {* 1 {* x y}}}
z	2
y	4
x	3
foo	{fun {x} {+ x {+ y z}}}
z	3
y	3
x	4

Tabla 4: Ambiente final

2. Sea una función f definida en el lenguaje de programación Racket como `number -> number`. Explica con tus propias palabras por qué si siempre:

$$2 * (f\ x) == (f\ x) + (f\ x)$$

entonces es un ejemplo que muestra la transparencia referencial de f.

Solución: Tengamos en cuenta que transparencia referencial puede ser definida como *reemplazar iguales por iguales* (como por ejemplo, reemplazar $2 + 2$ por 4). De esta forma, si siempre se cumple que

$$2 * (f\ x) == (f\ x) + (f\ x)$$

entonces éste un ejemplo de transparencia referencial, pues los valores finales son equivalentes y así es posible reemplazar iguales por iguales.

3. Transforma la siguiente función usando recursión de cola, y agrega los registros de activación de la llamada a función de `(division 4 2)`.

```
(define division
  (lambda (n m)
    (if (= n 0)
        1
        (+ 1 (division (- n m) m)))))
```

Solución: La función `division` puede ser optimizada usando recursión de cola de la siguiente manera

```
(define division-tail
  (local ((define sos
            (lambda (n m acc)
              (if (= n 0)
                  acc
                  (sos (- n m) m (+ 1 acc))))))
    (lambda (n m)
      (sos n m 1))))

(define division
  division-tail)
```

Por otro lado, los registros de activación de la llamada a función de `(division 4 2)` son:

Entra `(division 4 2)`

```
division-tail 4 2
  division-tail
      4 2
    division
```

Entra `(division-tail 4 2)`

```
              (sos 4 2 1)
(local ((define sos (lambda (n m acc)
           (if (= n 0) acc
    (sos (- n m) m (+ 1 acc))))))
     (lambda (n m) (sos n m 1)))
                4 2
           division-tail
```

Entra/Sale `(sos 4 2 1)`

```
(sos 2 2 2)
(if (= n 0) acc
(sos (- n m) m (+ 1 acc)))
4 2 1
sos
```

Entra/Sale `(sos 2 2 2)`

```
(sos 0 2 3)
(if (= n 0) acc
(sos (- n m) m (+ 1 acc)))
2 2 2
sos
```

Entra/Sale `(sos 0 2 3)`

```
3
(if (= n 0) acc
(sos (- n m) m (+ 1 acc)))
0 2 3
sos
```

Donde finalmente obtenemos el valor de 3.

4. Dentro del Cálculo Lambda, evalúa cada una de las siguientes expresiones usando $\beta-$reducciones. Si alguna tiene forma normal, especifícala.

   - $(\lambda x.x)(\lambda x.xxx)$
     Solución:

     $$\begin{aligned}(\lambda x.x)(\lambda x.xxx) &\to_\beta x[x := (\lambda x.xxx)] \\ &\to_\beta \lambda x.xxx\end{aligned}$$

     Como la expresión $\lambda x.xxx$ ya no puede reducirse más mediante $\beta-$reducciones, entonces ya se encuentra en Forma Normal.
   - $(\lambda x.(\lambda y.yxw)\ z)\ u$
     Solución:

     $$\begin{aligned}(\lambda x.(\lambda y.yxw)\ z)\ u &\to_\beta (\lambda y.yxw[x := z])\ u \\ &\to_\beta (\lambda y.yzw)\ u \\ &\to_\beta yzw[y := u] \\ &\to_\beta uzw\end{aligned}$$

     Como la expresión uzw ya no puede reducirse más mediante $\beta-$reducciones, entonces ya se encuentra en Forma Normal.
   - $(\lambda x.\lambda y.\lambda z.x)(yz)$
     Solución:

     $$\begin{aligned}(\lambda x.\lambda y.\lambda z.x)(yz) &\to_\beta \lambda y.\lambda z.x[x := (yz)] \\ &\to_\beta \lambda y.\lambda z.yz\end{aligned}$$

     Como la expresión $\lambda y.\lambda z.yz$ ya no puede reducirse más mediante $\beta-$reducciones, entonces ya se encuentra en Forma Normal.

5. Define el combinador Y de manera formal. Da un ejemplo de uso de éste.

   Solución: El combinador Y se define formalmente como sigue

   $$Y =_{def} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

   La siguiente expresión define la función `fibonacci` usando la primitiva `let` de Racket.

```
(let ([fib (lambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))]))
```

   Para usar el combinador Y debemos adaptar nuestra definición de tal forma que la función reciba una función parámetro.

```
(let ([fib (lambda (fib)
             (lambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))])
  (fib 3))
```

   De esta manera, podemos definir Y mediante un identificador y aplicarlo a `fib`. Usamos `let*` para facilitar la escritura.

```
(let* ([Y (lambda (f) ((lambda (x) (f (x x))) (lambda (x) (f (x x)))))]
       [fib (Y (lambda (fib)
                 (lambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))))])
  (fib 3))
```

   Al ejecutar esta expresión se cicla el programa, pues siempre estamos generando una nueva aplicación de `fib`.

6. Da un ejemplo en Racket donde uses la estructura de cajas y expongas el concepto de estado y SPS. En el ejemplo se debe reflejar el cambio de estado de una variable definida como una caja, cuyo valor dentro de la caja sea inicialmente 0 (cero) y termine con valor de 2, teniéndo un valor de 1 de forma intermedia.

   Solución: Definimos la expresión `expr` como

```
'{with {b {newbox 0}}
   {seqn {setbox b {+ 1 {openbox b}}}
         {setbox b {+ 1 {openbox b}}}
         {openbox b}}}
```

   Para interpretar un programa, llamamos a la función `interp` con un ambiente y un Store (los cuales vimos en clase).

```
(vxs-value (interp (parse expr) (mtSub) (mtSto)))
```

   Lo cual nos regresa como resultado `(num 2)`. Notemos que inicialmente la caja tiene un valor de 0, luego adquiere un valor de 1 al ejecutarse la primer línea del `seqn` y finalmente toma un valor de 2 al ejecutarse la segunda líneal del `seqn`. Como la última línea de éste es la que se regresa, entonces obtenemos el valor actual de la caja, el cual es 2.