Lenguajes de Programación, 2019-2
Nota de laboratorio 8
Evaluación perezosa

Manuel Soto Romero

1 de abril de 2019

Facultad de Ciencias, UNAM

"La evaluación perezosa es una estrategia de evaluación dónde los parámetros en la llamada de un procedimiento no son evaluados hasta que sean requeridos en el cuerpo del procedimiento."
— Matthias Felleisen

En esta nota se revisa la implementación de un intérprete que use evaluación perezosa y de discuten las implicaciones con respecto al alcance y al proceso de evaluación mediante puntos estrictos.

1. Gramática para condicionales

Se extiende la gramática del lenguaje `LDP` añadiendo el condicional `if0` que verifica si la expresión recibida como primer argumento es cero, en cuyo caso regresa el resultado de interpretar el segundo argumento o en caso contrario, la interpretación del tercer argumento. La nueva gramática se presenta a continuación:

```
<expr> ::= <id>
         | <num>
         | {if0 <expr> <expr> <expr>}
         | {<binop> <expr> <expr>}
         | {with {<id> <expr>} <expr>}
         | {fun {<id>} <expr>}
         | {<expr> <expr>}

<id> ::= A | B | ... | a | ... | z | aa | ...
       (Cualquier combinación de caracteres alfanumérica)

<num> ::= ... | -1 | 0 | 1 | ... | 2.5
        (Cualquier número entero o flotante)

<binop> ::= + | - | * | /
```

Observación 1. Recordar que `LDP` es un lenguaje de programación muy parecido sintáctica y semánticamente a Racket, el metalenguaje que se usará a lo largo de estas notas. Para diferenciar entre un lenguaje y otro se hace uso de llaves en lugar de corchetes y se usa el símbolo (λ) como apuntador (*prompt*) de `LDP`.

1.1. Tipos de datos básicos

El único tipo de dato de LDP son los números y son admitidos números enteros, flotantes, racionales y complejos. Además, los números pueden ser de cualquier longitud.

```
(λ) -1
-1
(λ) 1729
1729
(λ) 18.35
18.35
(λ) 1/2
1/2
(λ) 5+3i
5+3i
(λ) 17283982937489234750
17283982937489234750
```

1.2. Funciones predefinidas

1.2.1. Funciones aritméticas

Para operar con números se tienen las siguientes funciones:

- La función suma (+) permite realizar sumas de forma binaria.

```
(λ) {+ 1 2}
3
(λ) {+ {+ 1 2} {+ 3 4}}
10
```

- La función resta (-) permite realizar restas de forma binaria.

```
(λ) {- 5.0 3.5}
1.5
(λ) {- {+ 1 2} {- 3 2}}
2
```

- La función producto (*) permite realizar multiplicaciones de forma binaria.

```
(λ) {* 2.5 4.0}
10.0
(λ) {* {+ 1 4} {- 4 2}}
10
```

- La función división (/) permite realizar divisiones de forma binaria.

```
(λ) {/ 10 4}
2.5
(λ) {/ {+ 8 {- 4 2}} {* 2 2}}
2.5
```

1.3. Condicionales

El único tipo de condicional en `LDP` es `if0`.

```
(λ) {if0 {- 5 5} 3 2}
3
(λ) {if0 {- {* 5 2} 3} 17 29}
29
```

1.4. Asignaciones locales

Para poder asignar variables con un alcance restringido, en `LDP` se tiene la primitiva `with`.

```
(λ) {with {x 5} x}
5
(λ) {with {x 5}
       {with {y {+ x x}}
           {* x y}}}
50
```

1.5. Funciones anónimas

Para definir funciones anónimas (lambdas) se usa la primitiva `fun`, la aplicación de una función consiste en poner entre corchetes la función y el argumento con el cual se desea aplicar.

```
(λ) {fun {x} x}
#<function>
(λ) {{fun {x} x} 2}
2
(λ) {with {foo {fun {x} {+ x 2}}}
      {foo 2}}
4
```

2. Análisis sintáctico

Para representar la sintaxis abstracta, se define, en el Listado de código 1, el tipo de dato `CFWAE/L`.

```
1  ;; Tipo de dato para los ASA de LDP.
2  (define-type CFWAEL
3     [idS    (i symbol?)]
4     [numS   (n number?)]
5     [if0S   (codition CFWAE/L?) (then-expr CFWAE/L?) else-expr CFWAE/L?)]
6     [binopS (f procedure?) (lhs CFWAE/L?) (rhs CFWAE/L?)]
7     [withS  (id symbol?) (value CFWAE/L?) (body CFWAE/L?)]
8     [funS   (param symbol?) (body CFWAE/L?))]
9     [appS   (fun-expr CFWAE/L?) (arg CFWAE/L?)])
```

Listado de código 1: Tipo de dato CFWAE/L

En el Listado de código 2 se implementa la función `parse` que realiza el análisis sintáctico correspondiente.

```
1  ;; Analizador sintáctico de LDP.
2  ;; parse: s-expression -> CFWAE/L
3  (define (parse sexp)
4     (match sexp
5        [(? symbol?) (idS sexp)]
6        [(? number?) (numS sexp)]
7        [(list 'if0 codition then-expr else-expr)
8           (if0s (parse codition) (parse then-expr) (parse else-expr))]
9        [(list 'with (list id value) body)
10          (withS id (parse value) (parse body))]
```

```
      [(list 'fun (list param) body)
         (funS param (parse body))]
      [(list op izq der)
         (binopS (elige op) (parse izq) (parse der))]
      [(list fun-expr arg)
         (appS (parse fun-expr) (parse arg))]))
```

Listado de código 2: Analizador sintáctico de LDP

Se añade un nuevo caso a la función implementada en la Nota de Laboratorio 6:

Condicional `if0`

El condicional `if0` se representa mediante listas (con cuatro elementos que siempre inician con el símbolo `'if0`) y se transforma a expresiones de tipo `CFWAE/L` mediante el constructor `if0S`. El constructor `if0S`, recibe:

- Una condición, una caso `then` y caso `else` de tipo `CFWAE/L`, que deben procesarse recursivamente para obtener una expresión del mismo tipo.

3. Azúcar sintáctica

En el Listado de código 3 se implementa la función `desugar` que realiza el desendulzamiento de expresiones, esta función, recibe datos de tipo `CFWAE/L` y regresa datos de tipo `CFAE/L`.

```
;; Tipo de dato para los ASA de LDP sin azúcar sintáctica.
(define-type CFAE/L
   [id    (i symbol?)]
   [num   (n number?)]
   [if0   (condition CFAE/L?) (then-expr CFAE/L) (else-expr CFAE/L?)]
   [binop (f procedure?) (lhs CFAE/L?) (rhs CFAE/L?)]
   [fun   (param symbol?) (body CFAE/L?))]
   [app   (fun-expr CFAE/L?) (arg CFAE/L?)])

;; Desendulzador de LDP.
;; desugar: FWAE -> FAE
(define (desugar expr)
   (match expr
      [(idS i) (id i)]
      [(numS n) (num n)]
        [(if0s condition then-expr else-expr)
         (if0 (desugar condition)
              (desugar then-expr)
              (desugar else-expr))]
```

```
20        [(binopS f izq der) (binop f (desugar izq) (desugar der))]
21        [(withS id value body)
22           (app (fun id (desugar body)) (desugar value))]
23        [(funS param body)
24           (fun param (desugar body))]
25        [(appS fun-expr arg)
26           (app (desugar fun-expr) (desugar arg))]))
```

Listado de código 3: Desendulzamiento de expresiones

Las líneas 16 a 19 muestran el desendulzamiento de expresiones `if0`.

4. Análisis semántico

A continuación se revisa el comportamiento de las nueva primitiva `if0` del lenguaje.

Condicional `if0` El condicional `if0` verifica que el resultado de evaluar la condición sea cero en cuyo caso se regresa el resultado de evaluar el primer argumento y en caso contrario, se evalúa el segundo argumento.

```
(λ) {if0 {+ 10 2} 2 3}
3
(λ) {if0 {- 10 10} 2 3}
2
```

4.1. Una primera versión de `interp`

En esta nota, el intérprete para LDP, usa un régimen de evaluación perezosa. En un principio, lo único que debe modificarse en el intérprete es el valor que ingresa al ambiente, en este caso se debe introducir el argumento de las funciones sin evaluar, tal y como se muestra en la línea 16 del listado de código 4.

```
1  ;; Interpretación de expresiones de CFAE/L.
2  ;; interp: CFAE/L Env -> CFAE/L-Value
3  (define (interp env expr)
4     (match expr
5        [(id i) (lookup i env)]
6        [(num n) (numV n)]
7        [(if0 condition then-expr else-expr)
8           (if (zero? (interp condition env))
9               (interp then-expr env)
```

```
10              (interp else-expr env))]
11       [(binop f izq der)
12          (numV (f (numV-n (interp izq env))
13                   (numV-n (interp der env))))]
14       [(fun param body)
15          (closureV param body env)]
16       [(app fun-expr arg)
17          (let ([fun-val (interp fun-expr env)])
18             (interp (closureV-body fun-val)
19                     (aSub (closureV-param fun-val)
20                           arg
21                           (closureV-env fun-val))))]))
```

Listado de código 4: Primera versión del intérprete perezoso

A continuación se revisa la interpretación de la siguiente expresión, usando esta nueva versión de `interp`:

```
{with {a 3}
   {with {b {+ a a}}
      {with {a 4}
         b}}}
```

En sintaxis abstracta, sin azúcar sintáctica:

```
(app (fun 'a
          (app (fun 'b
                    (app (fun 'a
                              (id 'b))
                         (num 4))
               (binop + (id 'a) (id 'a))))
     (num 3))
```

Ejecución:

1. Se ejecuta la llamada:

```
(interp
   (app (fun 'a
             (app (fun 'b
                       (app (fun 'a
                                 (id 'b))
                            (num 4))
                  (binop + (id 'a) (id 'a))))
          (num 3))
       (mtSub))
```

2. Se obtiene una nueva llamada, modificando el ambiente actual y reduciendo la expresión:

```
(interp
   (app (fun 'b
             (app (fun 'a
                       (id 'b))
                  (num 4))
        (binop + (id 'a) (id 'a))))
   (aSub 'a
         (num 3)
         (mtSub)))
```

3. Se repite el proceso:

```
(interp
   (app (fun 'a
             (id 'b))
        (num 4))
   (aSub 'b
         (binop + (id 'a) (id 'a))
         (aSub 'a
               (num 3)
               (mtSub)))
```

4. Una última reducción

```
(interp
   (id 'b)

   (aSub 'a
         (num 4)
         (aSub 'b
                (binop + (id 'a) (id 'a))
                (aSub 'a
                      (num 3)
                      (mtSub))))
```

   El ambiente final tiene la forma:

'a	(num 4)
'b	(binop + (id 'a) (id 'a))
'a	(num 3)

5. Se realiza la búsqueda correspondiente mediante la función `lookup`:

```
(lookup
   'b

   (aSub 'a
         (num 4)
         (aSub 'b
               (binop + (id 'a) (id 'a))
               (aSub 'a
                     (num 3)
                     (mtSub))))
```

Se obtiene `(binop + (id 'a) (id 'a))`

6. Se procesa ahora con la búsqueda de `'a` en el ambiente actual.

```
(lookup
   'a

   (aSub 'a
         (num 4)
         (aSub 'b
               (binop + (id 'a) (id 'a))
               (aSub 'a
                     (num 3)
                     (mtSub))))
```

Se obtiene `(num 4)`, con lo cual la expresión resultante es `(binop + (num 4) (num 4))`. De esta forma se aprecia que el intérprete vuelve a usar alcance dinámico.

Observación 2. En realidad, la ejecución termina en el paso 5, pues la interpretación de identificadores es un caso base de la función `interp`. Para poder continuar con la ejecución como se muestra en el paso 6, debe forzarse la evaluación de expresiones, esto se conoce como *punto* estricto y se define más adelante en esta nota.

4.2. Alcance estático en evaluación perezosa

El alcance dinámico se obtuvo debido a que se perdió el ambiente donde fueron definidas las expresiones que ingresan al evaluar las aplicaciones de función. Es decir, en versiones anteriores de `interp`, la expresión que se introducía en el ambiente era:

```
(interp arg env)
```

Sin embargo, esta nueva versión, simplemente introduce:

arg

Con lo cual, para implementar alcance estático en esta nueva versión de `interp`, se debe capturar o *encerrar* en ambiente actual con el argumento que está ingresando al ambiente. Esto se puede realizar mediante lo que se conoce como *cerradura de expresión*[1]. Una cerradura de expresión almacena una expresión y la asocia con un valor. El Listado de código 5 muestra la modificación al tipo de dato `CFAE/L-Value` que incluye cerraduras de expresión.

```
;; Tipos de datos devueltos por interp.
(define-type CFAE/L-Value
   [numV     (n number?)]
   [closureV (param symbol?) (body CFAE/L?) (env Env?)]
   [exprV    (expr CFAE/L?) (env Env?)])
```

Listado de código 5: Adición de cerraduras de expresión

De esta forma, en el Listado de código 6 se muestra una nueva versión de la función interp que hace uso de estas cerraduras para postergar la evaluación de expresiones. El alcance vuelve a ser estático pues las expresiones se evalúan en el ambiente donde fueron definidas.

```
;; Interpretación de expresiones de CFAE/L.
;; interp: CFAE/L Env -> CFAE/L-Value
(define (interp env expr)
   (match expr
      [(id i) (lookup i env)]
      [(num n) (numV n)]
      [(if0 condition then-expr else-expr)
         (if (zero? (interp condition env))
             (interp then-expr env)
             (interp else-expr env))]
      [(binop f izq der)
         (numV (f (numV-n (interp izq env))
                  (numV-n (interp der env))))]
      [(fun param body)
         (closureV param body env)]
      [(app fun-expr arg)
         (let ([fun-val (interp fun-expr env)])
            (interp (closureV-body fun-val)
                    (aSub (closureV-param fun-val)
                          (exprV arg env)
                          (closureV-env fun-val))))]))
```

Listado de código 6: Segunda versión del intérprete perezoso

[1] *Closure expressions.*

4.3. Puntos estrictos

"A los puntos donde la implementación de un lenguaje perezoso fuerza la reducción de una expresión a un valor (si existe) se les llama puntos estrictos del lenguaje"
— Shriram Krishnamurthi

No es difícil ver que al evaluar la siguiente expresión con el uso de cerraduras de expresión:

```
{with {a {+ 3 4}}
   {with {b {+ a a}}
      {+ b a}}}
```

se obtiene un error, pues al tratar de interpretar la operación binaria:

```
(binop + (id 'b) (id 'a))
```

con el siguiente ambiente:

γ

'b	(exprV (binop + (id 'a) (id 'a)) γ)
'a	(exprV (binop + (num 3) (num 4)) (mtSub))

se intenta realizar la siguiente operación:

```
(+ (numV-n (exprV (binop + (id 'a) (id 'a)) γ)) (numV-n (exprV (binop + (num 3) (num 4)) (mtSub))))
```

Sin embargo, la función `numV-n` arroja un error pues se trata de extraer el valor numérico de una cerradura de expresión y no es posible. Es necesario que se evalúe la expresión capturada en la cerradura con el ambiente correspondiente antes de poder aplicar la operación binaria, es decir, debe forzarse la evaluación y por lo tanto se puede concluir que los operandos de una operación binaria son un *punto estricto*.

Otros puntos estrictos de `LDP`, que pueden encontrarse siguiendo el mismo análisis, son:

- La condición en las expresiones `if0`, es decir, el primer argumento del constructor `if0`.
- La función en las aplicaciones de función, es decir, el primer argumento del constructor `app`.

Para forzar la evaluación de expresiones, debe realizarse una interpretación exhaustiva hasta llegar a un valor atómico, esto es, que no sea de tipo cerradura de expresión (`exprV`). El Listado de código 7 muestran la implementación de la función `strict` que reduce expresiones a un valor.

```
;; Reduce expresiones a un valor concreto.
;; strict: FAE-Value? -> FAE-Value?
(define (strict expr)
   (match expr
       [(exprV e env) (strict (interp e env))]
       [else expr]))
```

Listado de código 7: Aplicación de puntos estrictos

De esta forma, el Listado de código 8, muestra el uso de la función `strict` para forzar la evaluación de expresiones en los puntos estrictos correspondientes.

```
;; Interpretación de expresiones de CFAE/L.
;; interp: CFAE/L Env -> CFAE/L-Value
(define (interp env expr)
   (match expr
      [(id i) (lookup i env)]
      [(num n) (numV n)]
      [(if0 codition then-expr else-expr)
         (if (zero? (strict (interp condition env)))
             (interp then-expr env)
             (interp else-expr env))]
      [(binop f izq der)
         (numV (f (numV-n (strict (interp izq env)))
                  (numV-n (strict (interp der env)))))]
      [(fun param body)
         (closureV param body env)]
      [(app fun-expr arg)
         (let ([fun-val (strict (interp fun-expr env))])
            (interp (closureV-body fun-val)
                    (aSub (closureV-param fun-val)
                          (exprV arg env)
                          (closureV-env fun-val))))]))
```

Listado de código 8: Tercera versión del intérprete perezoso