Lenguajes de Programación, 2019-2
Nota de laboratorio 7
Alcance

Manuel Soto Romero

25 de marzo de 2019

Facultad de Ciencias, UNAM

El alcance que tome una variable puede repercutir en el resultado de evaluación de una expresión de manera significativa. En general existen dos tipos de alcance: *estático* y *dinámico*. El primero preserva el valor de las variables al momento de declarar una expresión en el lenguaje, mientras que el segundo toma el valor actual de las variables. En esta nota se analiza la función `interp` implementada en la Nota de Laboratorio 6 con el fin de optimizarla y estudiar su funcionamiento mediante los dos tipos de alcance.

1. Problemas con el algoritmo de sustitución

Dada la siguiente expresión en sintaxis concreta:

```
{with {a 3}
   {with {foo {fun {x} {+ x a}}}
      {with {a 4}
         {foo 3}}}}
```

Los siguientes pasos permiten el valor final de la expresión dada:

Análisis léxico El primer paso consiste en descomponer la cadena en lexemas, para lo cual se hace uso de la primitiva `quote` de Racket, con lo cual la expresión final queda como:

```
'{with {a 3}
    {with {foo {fun {x} {+ x a}}}
       {with {a 4}
          {foo 3}}}}
```

Análisis sintáctico A partir de la lista de lexemas, se construye una representación intermedia que permita a la computadora entender y procesar la expresión. En este caso se usan Árboles de Sintaxis Abstracta (ASA), representados mediante el tipo de dato `FWAE`, con lo cual se obtiene la siguiente expresión después de ejecutar la función `parse` correspondiente:

```
(withS 'a (numS 3)
    (withS 'foo (funS 'x (binopS + (idS 'x) (idS 'a)))
        (withS 'a (numS 4)
            (appS (idS 'foo) (numS 3)))))
```

Desendulzamiento Una vez construido el ASA, se eliminan aquellas expresiones con azúcar sintáctica, en este caso, como se definió en la Nota de laboratorio 6, las expresiones `with`, son una versión endulzada de aplicaciones a función, con lo cual la expresión resultante queda como sigue después de ejecutar la función `desugar` correspondiente:

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

Análisis semántico Para realizar el análisis semántico correspondiente, se usa la función `interp1` definida en la Nota de laboratorio 6, que se muestra en el Listado de código 1.

```
1  ;; Interpretación de expresiones de FAE.
2  ;; interp: FAE -> FAE
3  (define (interp1 expr)
4     (match expr
5        [(id i) (error 'interp1 "Free ID")]
6        [(num n) expr]
7        [(binop f izq der)
8           (num (f (num-n (interp1 izq)) (num-n (interp1 der))))]
9        [(fun param body) expr]
10       [(app fun-expr arg)
11          (let ([fun-val (interp1 fun-expr)])
12             (interp (subst (fun-body fun-val)
13                            (fun-param fun-val)
14                            (interp1 arg))))])))
```

Listado de código 1: Interpretación mediante sustitución de variables

- Dada la aplicación de función anterior, se procede a realizar la sustitución del parámetro de la función, en el cuerpo de la misma por la interpretación del argumento. Esto es:

```
(subst
     (app (fun 'foo
          (app (fun 'a
               (app (id 'foo) (num 3))
          (num 4)
     (fun 'x (binop + (id 'x) (id 'a))))

     'a

     (num 3))
```

Obteniendo como resultado:

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

- Se tiene nuevamente una aplicación de función por lo que se repite el proceso, reduciendo la expresión. Esto es:

```
(subst
    (app (fun 'a
        (app (id 'foo) (num 3))
    (num 4)

    'foo

    (fun 'x (binop + (id 'x) (num 3))))
```

Obteniendo como resultado:

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

- Se repite el proceso, reduciendo la expresión. Esto es:

```
(subst
    (app (fun 'x (binop + (id 'x) (num 3)) (num 3))

    'a

    (num 3))
```

Obteniendo como resultado:

```
(app (fun 'x (binop + (id 'x) (num 3)) (num 3))
```

- Se realiza una última sustitución:

```
(subst
    (binop + (id 'x) (num 3))

    'x

    (num 3))
```

Obteniendo como resultado:

```
(binop + (num 3) (num 3)) => (num 6)
```

Para reducir la expresión a un resultado, el intérprete tuvo que aplicar el algoritmo de sustitución cuatro veces: una por cada aplicación de función. En general, si un programa tiene tamaño n (el número de nodos del ASA), entonces cada sustitución recorre el resto del programa una vez, haciendo que la complejidad para este intérprete sea $O(n^2)$ en el peor de los casos.

2. Interpretación mediante ambientes

Es posible mejorar esta complejidad mediante el uso de *ambientes*. Un *ambiente* es una estructura que permite almacenar parejas de la forma $(id, valor)$, con lo cual la sustitución de variables se vuelve una búsqueda de valores en el ambiente, dado un identificador. Si se usa una estructura de datos como una pila o lista, la búsqueda de valores, se vuelve lineal, $O(n)$, lo cual reduce significativamente la complejidad en el peor de los casos.

Para representar ambientes, se define, en el Listado de código 2, el tipo de dato `Env`. Es fácil notar que se usa una representación de lista: se tiene un constructor para el ambiente vacío (`mtSub`) y un constructor `aSub` para construir un ambiente formado por un primer elemento (una pareja $(id, valor)$) y el resto del ambiente.

```
1 ;; Representación de ambientes.
2 (define-type Env
3    [mtSub]
4    [aSub (id symbol?) (value FAE?) (rest-env Env?)])
```

Listado de código 2: Ambientes en forma de lista

De esta forma, la firma de la función `interp`, cambia a:

interp: FAE Env -> FAE

La función `interp` recibe un ambiente en el cual se buscará el valor de cada variable. El Listado de código 3 muestra esta implementación.

```
1 ;; Función que realiza la búsqueda de variables en el ambiente.
2 ;; lookup: symbol Env -> FAE
3 (define (lookup sub-id env)
4    (match env
5       [(mtSub) (error 'lookup "Free ID")]
6       [(aSub id value rest-env)
7          (if (symbol=? id sub-id)
```

```
8                value
9                (lookup sub-id rest-env))]))
10
11  ;; Interpretación de expresiones de FAE.
12  ;; interp2: FAE Env -> FAE
13  (define (interp2 env expr)
14     (match expr
15        [(id i) (lookup i env)]
16        [(num n) expr]
17        [(binop f izq der)
18           (num (f (num-n (interp2 izq env))
19                   (num-n (interp2 der env))))]
20        [(fun param body) expr]
21        [(app fun-expr arg)
22           (let ([fun-val (interp2 fun-expr env)])
23              (interp (fun-body fun-val)
24                      (aSub (fun-param fun-val)
25                            (interp arg)
26                            env)))]))
```

Listado de código 3: Interpretación mediante ambientes

En el Listado de código 3, la función `lookup` (líneas 3 a 7) se encarga de realizar la búsqueda de variables en el ambiente, si ésta no se encuentra en el mismo, se reporta un error indicando que se tiene una instancia libre. De esta forma cambian dos casos en el intérprete:

1. La interpretación de identificadores (línea 15), consiste en llamar a la función `lookup` con el símbolo correspondiente y dejar que esta función encuentre el valor asociado al mismo.
2. La interpretación de aplicaciones de función (líneas 21 a 26) consiste en interpretar el cuerpo de la función correspondiente en el ambiente formado por el parámetro de la función y el argumento de la aplicación de función sobre el ambiente actual.

A continuación se revisa la interpretación de la siguiente expresión, usando ambientes:

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

- Dada la aplicación de función anterior, se procede a realizar la interpretación del cuerpo de la función, con el ambiente actual, en este caso el vacío.

```
(interp
    (app (fun 'foo
        (app (fun 'a
            (app (id 'foo) (num 3))
        (num 4)
    (fun 'x (binop + (id 'x) (id 'a))))

    (aSub 'a (num 3) (mtSub))
```

- Se procede ahora, con el cuerpo de la nueva función y se construye un nuevo ambiente:

```
(interp
    (app (fun 'a
        (app (id 'foo) (num 3))
    (num 4))

    (aSub 'foo (fun 'x (binop + (id 'x) (id 'a)))
          (aSub 'a (num 3) (mtSub))))
```

- Se repite el proceso anterior:

```
(interp
    (app (id 'foo) (num 3))

    (aSub 'a (num 4)
          (aSub 'foo (fun 'x (binop + (id 'x) (id 'a)))
                (aSub 'a (num 3) (mtSub))))))
```

- Se ingresa la última entrada al ambiente, en este caso la interpretación se realiza con el cuerpo de la función `foo`.

```
(interp
    (binop + (id 'x) (id 'a))

    (aSub 'x (num 3)
          (aSub 'a (num 4)
                (aSub 'foo (fun 'x (binop + (id 'x) (id 'a)))
                      (aSub 'a (num 3) (mtSub)))))))
```

- Para interpretar la operación, se realiza la interpretación del lado izquierdo y el derecho. Se realizan dos búsquedas en el ambiente mediante la función `lookup`. La primera búsqueda, `(id 'x)`, regresa `(num 3)`, mientras que la segunda búsqueda, `(id 'a)`, obtiene `(num 4)`, por lo tanto, se tiene:

```
(num (+ (num-n (num 3)) (num-n (num 4))))
```

  El resultado final es `(num 7)`.

Al analizar los resultados para esta expresión después de ejecutar los intérpretes `interp1` e `interp2`, se puede apreciar que se tienen dos resultados, 6 y 7 respectivamente. Esto se debe al alcance que toman las variables.

Definición 1. El *alcance* de una variable de es la región de un programa, donde ésta encuentra su valor.

Definición 2. Se dice que un lenguaje de programación usa *alcance estático*, cuando el alcance que toma una variable es el del ambiente construido al momento de definirla.

Definición 3. Se dice que un lenguaje de programación usa *alcance dinámico*, cuando el alcance que toma una variable es el del ambiente actual.

Ejemplo 1. La expresión:

```
{with {a 3}
   {with {foo {fun {x} {+ x a}}}
      {with {a 4}
         {foo 3}}}}
```

1. Evaluada con la función `interp1`, regresa 6, por lo tanto usa alcance estático. La variable `a` en el cuerpo de la función, alcanza su valor con la pareja `{a 3}`, es decir, la del ambiente construido cuando fue definida la función.
2. Evaluada con la función `interp2`, regresa 7, por lo tanto usa alcance dinámico. La variable `a` en el cuerpo de la función, alcanza su valor con la pareja `{a 4}`, es decir, la del ambiente actual.

□

3. Alcance estático mediante cerraduras

Para regresar el alcance estático al intérprete definido con ambientes, es necesario saber el valor es las variables al momento de definir funciones, esto se logra mediante una estructura llamada *cerradura*[1]. De esta formal, al momento de interpretar una función, se debe capturar el valor del ambiente y los datos de la función, es decir, su parámetro y cuerpo.

Para unificar el tipo de regreso de la función `interp`, se define el tipo de dato `FAE-Value`, que incluye un constructor para definir cerraduras. El Listado de código 4 muestra la definición de este tipo de dato.

[1] Del inglés *closure*.

```
1  ;; Definición del tipo FAE-Value
2  (define-type FAE-Value
3     [numV      (n number?)]
4     [closureV (param symbol?) (body FAE?) (env Env?)])
```

Listado de código 4: Definición de FAE-Value

También es necesario modificar el tipo de dato `Env`, como se muestra en el Listado de código 5, el valor que encierra cada ambiente es ahora de tipo `FAE-Value` (línea 4).

```
1  ;; Representación de ambientes.
2  (define-type Env
3     [mtSub]
4     [aSub (id symbol?) (value FAE-Value?) (rest-env Env?)])
```

Listado de código 5: Ambientes con el tipo FAE-Value

Finalmente, el Listado de código 6, muestra una tercera versión del intérprete usando ambientes y cerraduras, para optimizar la sustitución de variables y usando alcance estático.

```
1  ;; Interpretación de expresiones de FAE.
2  ;; interp3: FAE Env -> FAE-Value
3  (define (interp3 env expr)
4     (match expr
5        [(id i) (lookup i env)]
6        [(num n) (numV n)]
7        [(binop f izq der)
8           (numV (f (numV-n (interp3 izq env))
9                    (numV-n (interp3 der env))))]
10       [(fun param body)
11          (closureV param body env)]
12       [(app fun-expr arg)
13          (let ([fun-val (interp3 fun-expr env)])
14             (interp (closureV-body fun-val)
15                     (aSub (closureV-param fun-val)
16                           (interp arg)
17                           (closureV-env fun-val))))])))
```

Listado de código 6: Interpretación mediante ambientes y cerradura

Se pueden apreciar las siguientes modificaciones:

1. En la línea 6, se envuelve el número `n` en un constructor de tipo `numV`.
2. En las líneas 8 a 10, se reemplazan las apariciones de `num` por `numV`.

3. En la línea 11, las funciones regresan una cerradura que captura el ambiente actual, es decir, el ambiente donde ésta fue definida.

4. En las líneas 13 a 17, en lugar de obtener el cuerpo y parámetro directamente de la función, se obtiene ahora de la cerradura que devuelve la interpretación de la función.

   De la misma manera, el resto del ambiente que se construye deja de ser el ambiente actual y se sustituye por el ambiente de la cerradura.

A continuación se revisa la interpretación de la siguiente expresión, usando ambientes:

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

- Dada la aplicación de función anterior, se procede a realizar la interpretación del cuerpo de la función, con el ambiente actual, en este caso el vacío.

```
(interp
     (app (fun 'foo
          (app (fun 'a
               (app (id 'foo) (num 3))
          (num 4)
     (fun 'x (binop + (id 'x) (id 'a))))

     (aSub 'a (num 3) (mtSub))
```

- Se procede ahora, con el cuerpo de la nueva función y se construye un nuevo ambiente:

```
(interp
     (app (fun 'a
          (app (id 'foo) (num 3))
     (num 4))

     (aSub 'foo
           (closureV 'x
                     (binop + (id 'x) (id 'a))
                     (aSub 'a (num 3) (mtSub)))
           (aSub 'a (num 3) (mtSub))))
```

- Se repite el proceso anterior:

```
(interp
    (app (id 'foo) (num 3))

    (aSub 'a (num 4)
          (aSub 'foo
                (closureV 'x
                          (binop + (id 'x) (id 'a))
                          (aSub 'a (num 3) (mtSub)))
                (aSub 'a (num 3) (mtSub))))
```

- Se ingresa la última entrada al ambiente, en este caso la interpretación se realiza con el cuerpo de la función `foo`. y en el ambiente donde se definió la función.

```
(interp
    (binop + (id 'x) (id 'a))

    (aSub 'x (num 3)
          (aSub 'a (num 3) (mtSub))))
```

- Para interpretar la operación, se realiza la interpretación del lado izquierdo y el derecho. Se realizan dos búsquedas en el ambiente mediante la función `lookup`. La primera búsqueda, `(id 'x)`, regresa `(num 3)`, mientras que la segunda búsqueda, `(id 'a)`, obtiene `(num 3)`, por lo tanto, se tiene:

```
(numV (+ (numV-n (num 3)) (numV-n (num 3))))
```

  El resultado final es `(numV 6)`. Con lo cual se aprecia que el intérprete usa alcance estático.