Lenguajes de Programación, 2022-1
Nota de clase 09: Estado

Karla Ramírez Pulido Manuel Soto Romero

30 de noviembre de 2021
Facultad de Ciencias UNAM

En la Nota de Clase 8, revisamos el concepto de *continuación* como una forma de manipular el estado de un programa completo mediante el uso de funciones de orden superior y la técnica *continuation passing style*. En esta nota revisaremos el concepto de *estado* refiriéndonos a la mutación de variables y/o estructuras de datos, presente en la mayoría de lenguajes de programación que permite modificar y almacenar valores en memoria.

9.1. Estructuras de datos mutables

Para comprender el conceptos de estado, extenderemos nuestro lenguaje con nuevos constructores para mutar estructuras de datos. Los primeros tres definen operaciones con *cajas*. Una caja, como hemos visto anteriormente, es una estructura donde podemos almacenar un valor, abrir la caja para recuperar dicho valor y modificar el contenido de la misma. Además, definiremos un constructor que permite ejecutar bloques de código secuencial, es decir se ejecuta una instrucción después de otra, el resultado de la expresión será el de la última instrucción dentro del bloque. Llamamos a este lenguaje BFCWAE[1]:

```
<expr> ::= <id>
         | <num>
         | {+ <expr> <expr>}
         | {- <expr> <expr>}
         | {if0 <expr> <expr> <expr>}
         | {with {<id> <expr>} <expr>}
         | {fun {<id>} <expr>}
         | {<expr> <expr>}
         | {newbox <expr>}
         | {setbox <expr> <expr>}
         | {openbox <expr>}
         | {seqn <expr> <expr>}
```

Los constructores `newbox`, `setbox` y `openbox` son equivalentes a las funciones `box`, `set-box!` y `unbox` de Racket respectivamente, mientras que la primitiva `seqn` es equivalente a `begin`.

Por ejemplo, la siguiente expresión:

```
{with {b {newbox 1729}}
    {seqn {setbox b {+ 1 {openbox b}}}
          {openbox b}}}
```

Define una caja con el identificador `b` que almacena el número 1729, posteriormente muta la caja para que su contenido sea 1729 más uno y abre la caja, por lo tanto la expresión se evalúa a 1730.

[1] *Boxes, Functions, Conditionals, With and Arithmetic Expressios.*

9.1.1. Sintaxis abstracta

La sintaxis abstracta de este lenguaje se presenta mediante el tipo de dato `BCFAE` en Racket como se muestra en el Código 1.

```
(define-type BCFAE
   [id      (i symbol?)]
   [num     (n number?)]
   [add     (izq BCFAE?) (der BCFAE?)]
   [sub     (izq BCFAE?) (der BCFAE?)]
   [if0     (test-expr BCFAE?) (then-expr BCFAE?) (else-expr BCFAE?)]
   [fun     (param symbol?) (body BCFAE?)]
   [app     (fun-expr BCFAE?) (arg BCFAE?)]
   [newbox  (value BCFAE?)]
   [setbox  (box-expr BCFAE?) (value BCFAE?)]
   [openbox (box-expr BCFAE?)]
   [seqn    (expr1 BCFAE?) (expr2 BCFAE?)])
```

Código 1: Sintaxis abstracta de `BCFAE`

Por ejemplo para la expresión

```
{with {b {newbox 1729}}
    {seqn {setbox b {+ 1 {openbox b}}}
          {openbox b}}}
```

la sintaxis abstracta correspondiente es:

```
(with 'b (newbox (num 1729))
   (seqn (setbox (id 'b) (add (num 1) (openbox (id 'b))))
         (openbox (id 'b))))
```

9.1.2. Memoria

Para hacer que nuestras cajas sean mutables, necesitamos añadir una memoria donde se almacenen los valores contenidos en las cajas, para recuperar dicho valor debemos proveer una dirección que permita identificar la posición de la memoria en la que éstos fueron almacenados. Esta memoria usualmente es conocida como *heap* e interactúa directamente con los ambientes de evaluación que hemos usado en notas anteriores, a los cuales se les conoce como *stack*.

El Código 2 muestra la definición del tipo de dato `Store` que modela la memoria para nuestro intérprete.

```
(define-type Store
   [mtSto]
   [aSto (location number?) (value BCFAE-Value?) (store Store?)])
```

Código 2: Tipo de dato `Store`

De esta forma nuestros ambientes de evaluación también deben modificarse de forma que en lugar de relacionar identificadores con valores, asociarán identificadores con direcciones en memoria para recuperar los valores correspondientes. El Código 3 muestra esta definición:

```
(define-type Env
   [mtSub]
   [aSub (id symbol?) (location number?) (env Env?)])
```

Código 3: Tipo de dato `Env`

Por ejemplo, para la siguiente expresión:

```
{with {a 10}
    {with {foo {fun {x} {+ x 2}}}
        {with {b {newbox 20}}
            {+ {foo a} {openbox b}}}}}
```

se tiene el siguiente ambiente de evaluación y memoria (de forma gráfica):

Ambiente

b	2
foo	1
a	0

Memoria

3	20
2	3
1	{fun {x} {+ x 2}}
0	10

Notemos cómo en la dirección de memoria 2, no estamos almacenando el valor de la caja como tal, si no la caja propiamente, mientras que la caja almacena la dirección en memoria del valor que está almacenando. Esto se debe a que en realidad la dirección 2, está guardando una especie de *apuntador* a la caja. La caja al ser una estructura de datos, simplemente reserva un espacio de memoria, en este caso, la dirección 3. Podemos observar un comportamiento similar en el lenguaje de programación C.

Similarmente debe modificarse la función `lookup` que busca el valor de un identificador en el ambiente. En realidad, deben definirse dos funciones: una para buscar en el ambiente de evaluación y otra para buscar en la memoria. Dichas funciones se muestran en el Código 4.

```
(define (env-lookup sub-id env)
   (type-case Env env
      [mtSub () (error 'interp "Variable libre")]
      [aSub  (id location rest-env)
         (if (symbol=? id sub-id)
             location
             (env-lookup sub-id rest-env))]))

(define (sto-lookup loc sto)
   (type-case Store sto
      [mtSto () (error 'interp "Dirección de memoria no encontrada")]
      [aSto  (location value rest-sto)
```

```
        (if (equal? loc location)
            value
            (sto-lookup loc rest-sto))]))
```

Código 4: Búsquedas en ambiente y memoria

9.1.3. Interpretación

Usando los tipos de datos anteriores, modificamos el intérprete de forma que reciba como entrada una expresión, un ambiente y una memoria. Los tipos de datos devueltos por el intérprete serán, números, cerraduras y cajas. El Código 5 muestra la definición del tipo de dato `BCFAE-Value`. Las cajas almacenarán la dirección en memoria del tipo de dato que encapsulan.

```
(define-type BCFAE-Value
   [numV     (n number?)]
   [closureV (param symbol?) (body symbol?) (env Env?)]
   [boxV     (location number?)])
```

Código 5: Resultados del intérprete

El Código 6 muestra la integración de todos los tipos de datos y funciones que hemos revisado para interpretar expresiones de nuestro lenguaje. El intérprete recibe una expresión, un ambiente de evaluación y una memoria.

```
;; interp: BCFAE Env Store → BCFAE-Value
(define (interp expr env sto)
   (type-case BCFAE expr
      [id  (i)
         (sto-lookup (env-lookup i env) sto)]
      [num (n)
         (numV n)]
      [add (izq der)
         (numV (+ (numV-n (interp izq env sto))
                  (numV-n (interp der env sto))))]
      [sub (izq der)
         (numV (- (numV-n (interp izq env sto))
                  (numV-n (interp der env sto))))]
      [if0 (test-expr then-expr else-expr)
         (if (zero? (numV-n (interp test-expr env sto)))
             (interp then-expr env sto)
             (interp else-expr env sto))]
      ...))
```

Código 6: Un primer intérprete

Analicemos nuestro intérprete hasta este punto:

- Para interpretar un identificador es necesario primero buscar en el ambiente de evaluación el índice del identificador y a partir de éste buscar el valor correspondiente en la memoria.

- La interpretación de números se mantiene sin cambios.
- La interpretación de sumas y restas consiste en evaluar el lado izquierdo y derecho y aplicar la operación correspondiente. Las llamadas recursivas usan el ambiente y memoria originales.
- Para el condicional `if0`, se evalúa si la condición es igual a cero en cuyo caso devolvemos la evaluación de la rama *then* y en caso contrario la evaluación de la rama *else*. Al igual que en el caso anterior, las llamadas recursivas usan el ambiente y memoria originales.

Esta interpretación, pareciera ser correcta, sin embargo, tenemos que recordar que el estado de la memoria puede cambiar en cualquier punto de la ejecución del programa y debemos mantener esos cambios en todo momento. Por ejemplo, la siguiente expresión:

```
{with {b {newbox 8}}
     {if0 {seqn {setbox b 29}
                {openbox b}}
          17
          {openbox b}}}
```

debería a evaluarse a 29, pues dentro de la condición del `if0` hubo un cambio en la memoria que hace que la caja `b` tenga dicho valor. Sin embargo, la implementación del Código 6 arroja 8 como resultado. Es decir, no se utilizó el valor actualizado de `b` pues como lo mencionábamos anteriormente, la evaluación de las ramas *then* y *else* usan el ambiente y memoria originales.

Para lograr el comportamiento deseado, debemos pasar llamada tras llamada la memoria actualizada, llamamos a esta técnica *Store Passing Style*[2].

9.2. *Store Passing Style*

Para utilizar esta técnica es necesario que cada expresión que genere el intérprete conozca el estado final de la memoria cuando ésta fue evaluada. De esta forma, añadimos un nuevo tipo de dato `Value×Store`, éste se muestra en el Código 7 y asocia los tipos de datos `BCFAE-Value` y `Store`.

```
(define-type Value×Store
   [v×s (value BCFAE-Value?) (store Store?)])
```

Código 7: Tipo de dato `Value×Store`

Ahora, usando este tipo de dato, modificamos el intérprete para que cada subllamada, use la versión actualizada de la memoria. Analicemos el intérprete por partes.

Identificadores

Se debe buscar la dirección del identificador dentro del ambiente y a partir de ésta obtener el valor correspondiente en memoria. Una vez encontrado ese valor, lo asociamos con la memoria actual. Se usa esta memoria pues no hubo ningún cambio.

[2] *Estilo de paso de memoria*

```
[id  (i)
   (v×s (sto-lookup (env-lookup i env) sto) sto)]
```

Números

Únicamente se asocia el número resultante con la memoria actual.

```
[num  (n)
   (v×s (numV n) sto)]
```

Operaciones aritméticas

Este es el primer caso en el que se aprecia el uso de la técnica *store passing style*. Se debe evaluar el lado izquierdo y a partir de este, usar la memoria resultante para evaluar el lado derecho. El resultado final consiste en aplicar la operación correspondiente asociándola con la memoria del lado derecho pues ésta es la memoria final.

```
[add (izq der)
   (let* ([izq-res (interp izq env sto)]
          [izq-val (v×s-value izq-res)]
          [izq-sto (v×s-store izq-res)]
          [der-res (interp der env izq-sto)]
          [der-val (v×s-value der-res)]
          [der-sto (v×s-store der-res)])
      (v×s (numV (+ (numV-n izq-val) (numV-n der-val))) der-sto))]

[sub (izq der)
   (let* ([izq-res (interp izq env sto)]
          [izq-val (v×s-value izq-res)]
          [izq-sto (v×s-store izq-res)]
          [der-res (interp der env izq-sto)]
          [der-val (v×s-value der-res)]
          [der-sto (v×s-store der-res)])
      (v×s (numV (- (numV-n izq-val) (numV-n der-val))) der-sto))]
```

Condicional `if0`

Se aplica la técnica *store passing style* a través de la evaluación de la condición y dependiendo de el valor de dicha evaluación se interprete la rama *then* o *else* correspondiente pero usando la memoria generada por la condición.

```
[if0 (test-expr then-expr else-expr)
   (let* ([test-res (interp test-expr env sto)]
          [test-val (v×s-value test-res)]
          [test-sto (v×s-store test-res)])
      (if (zero? (numV-n test-val))
          (interp then-expr env test-sto)
          (interp else-expr env test-sto)))]
```

Funciones

Simplemente se asocia la cerradura asociada a la función con la memoria actual, pues no se realiza ningún cambio en la misma.

```
[fun (param body)
   (v×s (closureV param body env) sto)]
```

Aplicación de funciones

Este es el caso más interesante, pues es aquí donde se añaden registros en la memoria, misma que se pasará llamada tras llamada usando *store passing style*. Primero evaluamos la función para obtener la cerradura correspondiente y usando la memoria resultante evaluamos el argumento de la aplicación, mismo que será almacenado en la memoria.

```
[app (fun-expr arg)
   (let* ([fun-res (interp fun-expr env sto)]
          [fun-val (v×s-value fun-res)]
          [fun-sto (v×s-store fun-res)]
          [arg-res (inter arg env fun-sto)]
          [arg-val (v×s-value arg-res)]
          [arg-sto (v×s-store arg-res)]
          [new-loc (next-location)])
      (interp (closureV-body fun-val)
              (aSub (closureV-param fun-val)
                    new-loc
                    (closureV-env fun-val))
              (aSto new-loc
                    arg-val
                    arg-sto)))]
```

Para generar las direcciones de memoria, se usa la función `next-location` que calcula la siguiente dirección de memoria a agregar. Una definición de esta función se muestra en el Código 8.

```
(define last-location -1) ;; al inicio del intérprete

;; next-location: number
(define (next-location)
   (begin
      (set! last-location (add1 last-location))
      last-location))
```

Código 8: Un primer intérprete

Como podemos ver esta función depende por completo de la variable `last-location` con lo cual se pueden tener efectos secundarios. Se deja como ejercicio al lector modificar la definición de esta función de forma tal que no tenga dichos efectos. *Hint:* Debe modificarse también un fragmento del intérprete.

Operaciones con cajas

Las operaciones con cajas simplemente modifican, buscan y agregan registros dentro de la memoria. La creación de cajas con `newbox` genera un nuevo registro, por lo tanto, debemos usar la función `next-location` e interpretar el valor a guardar en la caja. Al crear una caja, el resultado debe ser de tipo `boxV`

```
[newbox (value)
   (let* ([value-res (interp value env sto)]
          [value-val (v×s-value value-res)]
          [value-sto (v×s-store value-res)]
          [new-loc (next-location)])
      (v×s (boxV new-loc) (aSto new-loc value-val value-sto)))]
```

Para modificar una caja con `setbox` primero debemos encontrar la dirección en memoria de la caja actual. Para ello interpretamos el `box-expr` correspondiente, mismo que se reduce a una búsqueda en la memoria. Una vez hecho esto, se realiza un proceso similar al de `newbox`, sin embargo, se usa la dirección en memoria de la caja original. Esta operación regresa el valor a guardar como resultado.

```
[setbox (box-expr value)
   (let* ([box-res (interp box-expr env sto)]
          [box-val (v×s-value box-res)]
          [box-sto (v×s-store box-res)]
          [value-res (interp value env box-sto)]
          [value-val (v×s-value value-res)]
          [value-sto (v×s-store value-res)])
      (v×s value-val (aSto (boxV-location box-val) value-val value-sto)))]
```

Finalmente, para obtener el valor de una caja, simplemente debemos interpretarla para obtener su dirección en memoria y a partir de esta, realizar la búsqueda correspondiente en la memoria.

```
[openbox (box-expr)
   (let* ([box-res (interp box-expr env sto)]
          [box-val (v×s-value box-res)]
          [box-sto (v×s-store box-res)])
      (v×s (sto-lookup (boxV-location box-val) box-sto) box-sto))]
```

Bloques de código secuencial

Finalmente, para ejecutar bloques de código secuencial con `seqn`, que en esta versión únicamente incluye dos instrucciones. En este caso simplemente se ejecuta la primera instrucciones y la memoria modificada se pasa a la segunda, misma que se devuelve como resultado. Notar que el valor devuelto por la primera expresión, no es de importancia, simplemente las modificaciones en memoria que realiza.

```
[seqn (expr1 expr2)
   (let* ([expr1-res (interp expr1 env sto)]
          [expr1-sto (v×s-store expr1-res)]
          [expr2-res (interp expr2 env expr1-sto)]
          [expr2-val (v×s-value expr2-res)]
          [expr2-sto (v×s-store expr2-res)])
      (v×s expr2-val expr2-sto))]
```

9.3. Evaluación de expresiones

Usando el intérprete de la sección anterior, veamos cómo evaluar algunas expresiones.

Ejemplo 9.1. Evaluar la siguiente expresión usando alcance estático y evaluación glotona.

```
{with {a 10}
   {with {foo {fun {x} {+ x 2}}}
      {with {b {newbox 20}}
         {+ {foo a} {openbox b}}}}}
```

(1) Inicializamos el ambiente y la memoria

```
{with {a 10}                                  →
   {with {foo {fun {x} {+ x 2}}}
      {with {b {newbox 20}}
         {+ {foo a} {openbox b}}}}}
```

b	2
foo	1
a	0

3	20
2	3
1	{fun {x} {+ x 2}}
0	10

(2) Evaluamos el cuerpo del `with` más anidado. Tenemos una suma, por lo tanto evaluamos el lado izquierdo que es una aplicación de función, por lo tanto buscamos `foo` en el ambiente. y de acuerdo a la dirección obtenida la buscamos en la memoria.

```
{+ {foo a} {openbox b}}
```

→

b	2
foo	1
a	0

→

3	20
2	3
1	{fun {x} {+ x 2}}
0	10

(3) Sustituimos `foo` con el valor encontrado y evaluamos su cuerpo extendiendo el ambiente y la memoria. Debemos buscar primero la definición de `a`. Notar que el ambiente añade elementos de acuerdo al alcance y la memoria agrega todo en el tope.

```
{+ {foo a} {openbox b}}
{+ {{fun {x} {+ x 2}} a} {openbox b}}
{+ {+ x 2} {openbox b}}
```

→

b	2
foo	1
x	4
a	0

→

4	10
3	20
2	3
1	{fun {x} {+ x 2}}
0	10

(4) Buscamos `x` en el ambiente y la memoria y aplicamos la operación correspondiente.

```
{+ {foo a} {openbox b}}
{+ {{fun {x} {+ x 2}} a} {openbox b}}
{+ {+ x 2} {openbox b}}                       →
{+ {+ 10 2} {openbox b}}
{+ 12 {openbox b}}
```

b	2
foo	1
x	4
a	0

→

4	10
3	20
2	3
1	{fun {x} {+ x 2}}
0	10

(5) Evaluamos el lado derecho de la suma que corresponde con la apertura de una caja. Para ello primero, buscamos el valor de `b` en el ambiente y memoria que corresponde con una caja que encapsula una dirección de memoria, misma que se busca en la memoria para obtener el valor.

```
{+ {foo a} {openbox b}}
{+ {{fun {x} {+ x 2}} a} {openbox b}}
{+ {+ x 2} {openbox b}}
{+ {+ 10 2} {openbox b}}
{+ 12 {openbox b}}
{+ 12 3}
{+ 12 20}
32
```

→

b	2
foo	1
x	4
a	0

	4	10
→	3	20
→	2	3
	1	{fun {x} {+ x 2}}
	0	10

□

Ejemplo 9.2. Evaluar la siguiente expresión usando alcance estático y evaluación glotona.

```
{with {b {newbox 8}}
   {if0 {seqn {setbox b 29}
              {openbox b}}
        17
        {openbox b}}}
```

(1) Inicializamos el ambiente y la memoria

```
{with {b {newbox 8}}
   {if0 {seqn {setbox b 29}
              {openbox b}}
        17
        {openbox b}}}
```

→

b	0

1	8
0	1

(2) Evaluamos el cuerpo del `with` más anidado. Tenemos un condicional `if0` por lo tanto evaluamos la condición, que corresponde a una secuencialización, por lo tanto ejecutamos la primera expresión que muta el valor de la caja. Notar que `setbox` sobreescribe el valor, añadiendo un registro con el mismo índice.

```
{if0 {seqn {setbox b 29}
           {openbox b}}
     17
     {openbox b}}
```

→

b	0

	1	29
	1	8
→	0	1

(3) Ejecutamos la segunda expresión que abre la caja `b`, la buscamos en el ambiente y memoria y devolvemos el valor que envuelve la caja. Notar que la búsqueda en la memoria se realiza desde el tope.

```
{if0 {seqn {setbox b 29}
           {openbox b}}
     17
     {openbox b}}
{if0 {seqn {setbox b 29}
           1}
     17
     {openbox b}}
{if0 29 17 {openbox b}}
```

→

b	0

→

	1	29
	1	8
→	0	1

(4) Dado que el valor obtenido de abrir la caja es distinto de cero, procedemos a ejecutar la rama `else`*. Que corresponde con abrir la caja, la buscamos en el ambiente y memoria y buscamos el valor que envuelve la caja en la memoria.*

```
{openbox b}
29
```

□

9.4. Ejercicios

9.4.1. El Lenguaje BCFWAE

1. Dada la sintaxis concreta del lenguaje BCFWAE y su sintaxis abstracta, construye una expresión que defina una función "doble" que dado un número calcule el doble del mismo. Posteriormente define una caja con el valor 1825, actualiza su valor usando la función doble y finalmente abre la caja. Muestra además su sintaxis abstracta.

9.4.2. *Store Passing Style*

1. Evalúa las siguientes expresiones usando *Store Passing Style*. Es necesario que muestres los ambientes y memorias finales en cada caso. Considera un régimen de evaluación glotón y usa alcance estático.

   (a)
   ```
   {with {a {fun {b} {fun {h} {/ {* b h} 2}}}}
         {with {b {box 10}}
            {with {h {box {20}}
                {with {r {box 1279}}
                    {seqn {setbox r {{a {openbox b}} {openbox h}}}
                          {openbox r}}}}}}
   ```

   (b)
   ```
   {with {ds {fun {n} {fun {m} {if0 m -1 {/ n m}}}}}
          {with {x {box 10}}
              {with {y 0}
                 {with {r {box 1729}}
                     {seqn {setbox r {{ds {openbox x} y}}
                           {openbox r}}}}}}
   ```

Referencias

[1] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation*, Primera edición, 2007. [En línea: **https://plai.org/**].