Práctica 6

Excepciones y Continuaciones en la máquina $\mathscr{K}$

Favio E. Miranda Perea (favio@ciencias.unam.mx)
Pablo G. González López (pablog@ciencias.unam.mx)

Miércoles 6 de noviembre de 2019

Fecha de entrega: Miércoles 20 de noviembre de 2019 a las 23:59:59.

1 La Máquina $\mathscr{K}$

Como ya se discutió en clase, la técnica para definir la semántica dinámica de un lenguaje que se ha usado hasta ahora es bastante conveniente para poder probar distintas propiedades de los lenguajes, sin embargo, resulta poco favorable para usarla como guía de la implementación.

Recordemos que su definición se da a través de un sistema de transiciones que utiliza reglas que determinan dónde aplicar la siguiente instrucción, pero sin especificar cómo encontrarla dentro de la expresión. Para hacer este proceso explícito, se introduce el mecanismo llamado pila de control, que registrará el trabajo que queda por hacer después de que se ejecuta una instrucción. Usando esta pila se elimina la necesidad de contar con premisas en las reglas de transición, de tal modo que el sistema de transiciones define una máquina abstracta cuyos pasos están determinados por la información que se encuentra en el estado, tal como lo haría cualquier computadora.

1.1 Marcos

Los marcos son esqueletos estructurales que registran los cómputos pendientes de manera explícita. Se definen del siguiente modo:

- Operadores binarios

  $\overline{op(-, e_2) \text{ marco}}$

  $\overline{op(v_1, -) \text{ marco}}$

- Operadores unarios

  $\overline{op(-) \text{ marco}}$

- Condicional booleano

  $\overline{if(-, e_1, e_2) \text{ marco}}$

- Instrucciónes con ligado (`let`, `handle`)

  $\overline{let(-, x, e) \text{ marco}}$

- Función (`fn`)

  $\overline{fn(x, -) \text{ marco}}$

El espacio marcado con un guión indica la posición correspondiente al valor devuelto en la evaluación actual, el cual corresponde al lugar donde se está llevando a cabo la evaluación.

Modelaremos esto del siguiente modo:

```
type Pending = ()

data Frame = SuccF Pending
           | PredF Pending
           | AddFL Pending Expr
           | AddFR Expr Pending
           | MulFL Pending Expr
           | MulFR Expr Pending
           ...
```

Implementa las siguientes funciones:

1. (1 punto) Crea una instancia de la clase `Show` para los marcos de acuerdo a la sintaxis descrita en la nota 11 del curso.

1.2 Estados, transiciones y pilas de control

Un estado s de la máquina $\mathscr{K}$ consiste de una pila de control k y una expresión cerrada e que toma alguna de las siguientes formas:

1. Una *evaluación* de la forma $k \triangleright e$ que corresponde a la evaluación de la expresión cerrada e sobre la pila de control k.

2. Un *retorno* de la forma $k \triangleleft e$, donde e es un valor, que corresponde a la evaluación de la pila de control k sobre la expresión cerrada e.

3. Una *propagación de error* de la forma $k \blacktriangleleft e$, donde e es una expresión de error, que corresponde a la propagación del error en la pila de control k.

Modelaremos esto del siguiente modo:

```
State = E (Memory, Stack, Expr) | R (Memory, Stack, Expr) | P (
    Memory, Stack, Expr)
```

Donde la pila de control será una lista que almacenará los marcos de las expresiones.

```
Stack = [Frame]
```

Finalmente, las transiciones se definen inductivamente del siguiente modo:

- Valores

  $$\frac{}{k \triangleright v \rightarrow_{\mathcal{K}} k \triangleleft v}$$

- Operadores binarios

  $$\frac{}{k \triangleright op(e_1, e_2) \rightarrow_{\mathcal{K}} k; op(-, e_2) \triangleright e_1}$$

  $$\frac{}{k; op(-, e_2) \triangleleft v \rightarrow_{\mathcal{K}} k; op(v, -) \triangleright e_2}$$

  $$\frac{}{k; op(v_1, -) \triangleleft v_2 \rightarrow_{\mathcal{K}} k \triangleleft op_p(v_1, v_2)}$$

  op_p es el operador primitivo.

- Operadores unarios

  $$\frac{}{k \triangleright op(e) \rightarrow_{\mathcal{K}} k; op(-) \triangleright e}$$

  $$\frac{}{k; op(-) \triangleleft v \rightarrow_{\mathcal{K}} k \triangleleft op_p(v)}$$

  op_p es el operador primitivo.

- Condicional booleano

  $$\frac{}{k \triangleright if(e_1, e_2, e_3) \rightarrow_{\mathcal{K}} k; if(-, e_2, e_3) \triangleright e_1}$$

  $$\frac{}{k; if(-, e_2, e_3) \triangleleft true \rightarrow_{\mathcal{K}} k \triangleright e_2}$$

  $$\frac{}{k; if(-, e_2, e_3) \triangleleft false \rightarrow_{\mathcal{K}} k \triangleright e_3}$$

- Instrucciónes con ligado (`let`, `handle`)

  $$\frac{}{k \triangleright op(e_1, x.e_2) \rightarrow_{\mathcal{K}} k; op(-, x.e_2) \triangleright e_1}$$

  $$\frac{}{k; op(-, x.e_2) \triangleleft v \rightarrow_{\mathcal{K}} k \triangleright e_2[x := v]}$$

Como podemos observar evaluar cualquier valor x es simplemente regresarlo. Para evaluar cualquier operador $op(e)$, agregamos el marco correspondiente a la pila de control y evaluamos la expresión e; cuando esta se regresa como e', regresamos $op(e')$ a la pila de control original.

2 Excepciones con valor

Las excepciones efectúan una transferencia de control del punto donde son lanzadas a un manejador que las encapsula. Esta transferencia interrumpe el flujo de control normal de un programa en respuesta a una condición inusual, pueden usarse para avisar de una condición de error o para indicar la necesidad de un manejo especial del programa en circunstancias inusuales o no esperadas. Estas circunstancias podrían tratarse de:

- Una división entre cero.
- Un índice de un arreglo fuera de rango.
- Un archivo no encontrado.
- O un sistema sin memoria.

Se podrían usar condicionales para revisar y procesar posibles errores, sin embargo usar excepciones es comúnmente más conveniente, particularmente porque la transferencia a un manejador es conceptualmente directa e inmediata, en lugar de indirecta a través de revisiones explícitas.

Implementaremos un tipo de excepciones que encapsularán un valor al ser lanzadas.

```
data Expr = ...
  | Raise Expr -- raise(e)
  | Handle Expr Identifier Expr -- handle(e1, x.e2)
```

El argumento de la expresión `Raise` se evaluará para determinar el valor que se pasará al manejador. La expresión `Handle` liga la variable x a la expresión manejadora (segunda expresión). El valor asociado de la excepción estará ligado en la expresión manejadora en caso de que se genere una excepción cuando se evalúa la primera expresión.

La pila de marcos se extiende para incluir `RaiseF Pending` y `HandleF Pending Identifier Expr`.

A continuación se listan las reglas que formalizan esto:

$$\frac{}{k \triangleright raise(e) \rightarrow_{\mathcal{K}} k; raise(-) \triangleright e}$$

$$\frac{}{k; raise(-) \triangleleft v \rightarrow_{\mathcal{K}} k \blacktriangleleft raise(v)}$$

$$\frac{}{k \triangleright handle(e_1, x.e_2) \rightarrow_{\mathcal{K}} k; handle(-, x.e_2) \triangleright e_1}$$

$$\frac{}{k; handle(-, x.e_2) \triangleleft v \rightarrow_{\mathcal{K}} k \triangleleft v}$$

$$\frac{f \neq handle(-, x.e_2)}{k; f \blacktriangleleft raise(v) \rightarrow_{\mathcal{K}} k \blacktriangleleft raise(v)}$$

$$\frac{}{k; handle(-, x.e_2) \blacktriangleleft raise(v) \rightarrow_{\mathcal{K}} k \triangleright e_2[x := v]}$$

3 Continuaciones

Las continuaciones constituyen una técnica la cual proporciona una manera simple y natural de modificar el flujo de evaluación de una evaluación en lenguajes funcionales. La idea básica detrás de las continuaciones es la de considerar a la pila de control de un programa como un valor el cual puede devolverse o pasarse como argumento a otra función.

```
data Expr = ...
  | LetCC Identifier Expr
  | Continue Expr Expr
  | Cont Stack
```

La evaluación de `LetCC x e` bajo una pila `P` causa la materialización de la pila en `Cont(P)` la cual se liga a `x` y se prosigue evaluando la expresión `e`.

Una expresión de tipo `Continue(e1, e2)` pasa el control a la pila materializada `e2` enviándole el valor `e1`. Para esto primero se evalúa `e1`. Una vez devuelto el primer argumento se procede a evaluar el segundo. Finalmente al devolver un valor `v2` a la pila con tope `ContinueF(Cont(P), -)` causa que la pila actual de control actual se abandola y la evaluación continúe devolviendo `v2` a la pila `P`. A continuación se listan las reglas que formalizan esto:

$$\frac{}{k \triangleright letCC(x.e) \rightarrow_{\mathcal{K}} k \triangleright e[x := cont(k)]}$$

$$\frac{}{k \triangleright continue(e_1, e_2) \rightarrow_{\mathcal{K}} k; continue(-, e_2) \triangleright e_1}$$

$$\frac{}{k; continue(-, e_2) \triangleleft v_1 \rightarrow_{\mathcal{K}} k; continue(v_1, -) \triangleright e_2}$$

$$\frac{}{k; continue(cont(k'), -) \triangleleft v_2 \rightarrow_{\mathcal{K}} k' \triangleleft v_2}$$

Extiende o adecúa las siguientes funciones:

1. (1 punto) Crea una instancia de la clase `Show` para los estados de acuerdo a la sintaxis descrita anteriormente.
2. (1 punto) Extiende las funciones `frVars`, `subst` y `alphaEq`.
3. (5 puntos) `eval1`. Recibe un estado de la máquina $\mathcal{K}$ y devuelve un paso de la transición. Es decir:

   `eval1 (E (m, s, e)) = E (m', s', e')` syss $(m, s) \triangleright e \rightarrow_{\mathcal{K}} (m', s') \triangleright e'$

`eval1 (R (m, s, e)) = R (m', s', e')` syss $(m,s) \triangleleft e \rightarrow_{\mathscr{K}} (m',s') \triangleleft e'$

`eval1 (E (m, s, e)) = R (m', s', e')` syss $(m,s) \triangleright e \rightarrow_{\mathscr{K}} (m',s') \triangleleft e'$

`eval1 (R (m, s, e)) = P (m', s', e')` syss $(m,s) \triangleleft e \rightarrow_{\mathscr{K}} (m',s') \blacktriangleleft e'$

```
eval1 :: State -> State
```

Ejemplo:

```
*Main> eval1 (E ([], [], Add (I 1) (I 2)))
E ([AddFL () (I 2)], I 1)
*Main> eval1 (R ([], [OrFR (B True) (), AndFL () (B
    False)],
B False))
R ([], [AndFL () (B False)], B True)
*Main> eval1 (R ([], [Succ ()], B False))
P ([], [], Raise (B False))
```

Nota: En caso de no poder evaluar la expresión se deberá propagar un error que contenta la expresión que lo causó.

4. (1 punto) `evals`. Recibe un estado de la máquina $\mathscr{K}$ y devuelve un estado derivado de evaluar varias veces hasta obtener la pila vacía. Es decir:

   `evals (E (m, s, e)) = R (m', [], e')` syss $(m,s) \triangleright e \rightarrow^*_{\mathscr{K}} (m',\emptyset) \triangleleft e'$

   ```
   evals :: State -> State
   ```

   Ejemplo:

   ```
   *Main> evals (E ([], [], Let "x" (I 2) (Mul (Add (I 1) (
       V "x")) (V "x"))))
   R ([], [], I 6)
   *Main> evals (E ([], [], Let "x" (B True) (If (V "x") (V
        "x") (B False))))
   R ([], [], B True)
   *Main> evals (E ([], [], Handle (Add (B True) (I 1))) "x
       " (V "x"))
   R ([], [], B True)
   ```

5. (1 punto) `eval`. Recibe una expresión `EAB`, la evalúa con la máquina $\mathscr{K}$ y devuelve un valor si, iniciando con la pila vacía esta devuelve un valor a la pila vacía. Es decir:

   `eval e = e'` syss $(\emptyset,\emptyset) \triangleright e \rightarrow^*_{\mathscr{K}} (m',\emptyset) \triangleleft e'$ y e' es un valor.

   En caso contrario manda un mensaje de error.

   Utiliza la función `evale`.

```
eval :: Expr -> Expr
```

Ejemplo:

```
*Main> eval (Let "x" (I 1) (If (Gt (V "x") (I 0)) (Eq (V
    "x") (I 0))
(B True)))
B False
*Main> eval (Not (I 3))
*** Exception: Error.
```

¡Suerte!