Práctica 5
Implementación de MiniC

Miguel Ángel Ordóñez Silis (417034052) Aldair Coronel Ruiz (315126831)

Facultad de Ciencias, UNAM

6 de Noviembre, 2019

1. Descripción del programa y ejecuciones

MiniC es un pequeño lenguaje imperativo que tiene como núcleo el lenguaje MinHs visto anteriormente, añadiendo valores y operadores que permiten a los programas tener efectos laterales de control y almacenamiento.

Paradigma Imperativo = Paradigma Funcional + Efectos Laterales

Representaremos a la memoria como una lista de celdas, definiendo cada celda como una dupla de dirección de memoria y valor.

```
type Address = Int

type Value = Expr

type Cell = (Address, Value)

type Memory = [Cell]
```

Implementamos las siguientes funciones en un módulo llamado **Memory**:

1) **newAddress.** Dada una memoria, genera una nueva dirección de memoria que no este contenida en esta.

```
*Prelude> :load BAE.Memory
[1 of 2] Compiling BAE.Sintax          ( BAE/Sintax.hs, interpreted )
[2 of 2] Compiling BAE.Memory          ( BAE/Memory.hs, interpreted )
Ok, two modules loaded.
*BAE.Memory> newAddress []
```

```
L 0
*BAE.Memory> newAddress [(0, B False), (2, I 9)]
L 1
*BAE.Memory> newAddress [(0, I 21), (1, Void), (2, I 12)]
L 3
*BAE.Memory> newAddress [(0, I 21), (1, Void), (2, I 12), (1,B True)]
*** Exception: Corrupted memory.
CallStack (from HasCallStack):
error, called at ./BAE/Memory.hs:26:25 in main:BAE.Memory
```

2) **access.** Dada una dirección de memoria, devuelve el valor contenido en la celta con tal dirección, en caso de no encontrarla debe devolver **Nothing**.

```
*BAE.Memory> access 3 []
Nothing
*BAE.Memory> access 1 [(0, B False), (2, I 9)]
Nothing
*BAE.Memory> access 2 [(0, I 21), (2, I 12), (1, Void)]
Just num[12]
*BAE.Memory> access 2 [(0, I 21), (0, B False), (3, Void), (2, I 12)]
*** Exception: Corrupted memory.
CallStack (from HasCallStack):
error, called at ./BAE/Memory.hs:66:21 in main:BAE.Memory
```

3) **update.** Dada una celda de memoria, actualiza el valor de esta misma en la memoria. En caso de no existir debe devolver **Nothing**.

```
*BAE.Memory> update (3, B True)[]
Nothing
*BAE.Memory> update (0, Succ (V "x")) [(0, B False), (2, I 9)]
*** Exception: Memory can only store values.
CallStack (from HasCallStack):
error, called at ./BAE/Memory.hs:82:35 in main:BAE.Memory
*BAE.Memory> update (0, Fn "x" (V "x")) [(0, I 21), (1, Void), (2, I 12)
Just [(0,fn(x.V[x])),(1,()),(2,num[12])]
*BAE.Memory> update (2, I 14) [(0, I 21), (2, Void), (2, I 12)]
*** Exception: Corrupted memory.
CallStack (from HasCallStack):
error, called at ./BAE/Memory.hs:75:21 in main:BAE.Memory
```

Ciclo while

El constructor correspondiente:

```
While Expr Expr
```

Implementamos / Extendimos las siguientes funciones:

1) **frVars.** Extiende esta función para las nuevas expresiones.

```
*BAE.Memory> :load BAE/Dynamic.hs
 [1 of 3] Compiling BAE.Sintax          ( BAE/Sintax.hs, interpreted )
 [2 of 3] Compiling BAE.Memory          ( BAE/Memory.hs, interpreted )
 [3 of 3] Compiling BAE.Dynamic         ( BAE/Dynamic.hs, interpreted )
Ok, three modules loaded.
*BAE.Dynamic> frVars (Add (V "x") (I 5))
["x"]
*BAE.Dynamic> frVars (Assig (L 2) (Add (I 0) (V "z")))
["z"]
```

2) **subst.** Extiende esta función para las nuevas expresiones.

```
*BAE.Dynamic> subst (Add (V "x") (I 5)) ("x", I 10)
add(num[10],num[5])
BAE.Dynamic> subst (Assig (L 2) (Add (I 10) (V "z"))) ("z"*BAE.Dynamic>
let(num[1],x1.V[x1])
*, B False)
L 2 := add(num[10],bool[False])
```

3) **eval1.** Extiende esta función para que dada una memoria y una expresión, devuelva la reducción a un paso, es decir, **eval1 (m, e) = (m', e')** si y solo si $(m, e) \to (m', e')$.

```
*BAE.Dynamic> eval1 ([(0, B False)], (Add (I 1) (I 2)))
([(0,bool[False])],num[3])
*BAE.Dynamic> eval1 ([(0, B False)], (Let "x" (I 1) (Add (V "x") (I 2)))
([(0,bool[False])],add(num[1],num[2]))
*BAE.Dynamic> eval1 ([(0, B False)], Assig (L 0) (B True))
([(0,bool[True])],())
*BAE.Dynamic> eval1 ([], While (B True) (Add (I 1) (I 1)))
([], if(bool[True],add(num[1],num[1]);while(bool[True],add(num[1],num[1])
```

4) **evals.** Extiende esta función para que dada una memoria y una expresión, devuelva la expresión hasta que la reducción quede bloqueada, es decir, **evals (m, e) = (m', e')** si y solo si $(m, e) \to^* (m', e')$ y e' está bloqueado.

```
*BAE.Dynamic> evals ([], (Let "x" (Add (I 1) (I 2)) (Eq (V "x") (I 0))))
([],bool[False])
*BAE.Dynamic> evals ([], (Add (Mul (I 2) (I 6)) (B True)))
([],add(num[12],bool[True]))
*BAE.Dynamic> evals ([], Assig (Alloc (B False)) (Add (I 1) (I 9)))
([(0,num[10])],())
```

5) **evale.** Devuelve la evaluación de un programa tal que **evale e = e'** syss $e \rightarrow^* e'$ y e' es un valor. En caso de que e' no esa un valor deberá mostrar un mensaje de error particular del operador que lo causó

```
*BAE.Dynamic> evale (Add (Mul (I 2) (I 6)) (B True))
*** Exception: [Add] expects two Integer.
CallStack (from HasCallStack):
error, called at BAE/Dynamic.hs:272:49 in main:BAE.Dynamic
*BAE.Dynamic> evale (Or (Eq (Add (I 0) (I 0)) (I 0)) (Eq (I 1) (I 10)))
bool[True]
```

6) Por último, agregamos al directorio **demo** la implementación de varios programas que utilizan el ciclo *while*. Estos programas son:

- Division.miniC (División de enteros)
- Maximum-Common-Divisor.miniC (MCD)
- Predecessor.miniC (Función predecesor)

Los tres programas crean las funciones correspondientes y las ejecutan con ciertos valores que se escogieron arbitrariamente. Por ejemplo, el código de Division.miniC es el siguiente:

```
let division := fn m => fn n  => let q := alloc 0 in
                                     let r := alloc m in
                                         (while ((!r > n) | (!r = n)) do
                                               q ::= !q + 1;
                                               r ::= !r + (n * (-1))
                                         end);
                                         !q
                                     end
                                 end
in
        division $ 64 $ 4
end :: Integer
```

Cuya ejecución se ve así:

```
[mianorsi@Roach BAE]$ ./BAEi demo/Division.miniC
Program:
    let(fn(m.fn(n.let(alloc num[0],q.let(alloc V[m],r.while(
or(gt(!V[r],V[n]),eq(!V[r],V[n])),V[q] := add(!V[q],num[1]);V[r] :=
add(!V[r],mul(V[n],num[-1])));!V[q])))),division.app(app(V[division],
num[64]), num[4])) : Integer
Evaluation:
num[16]
```

Como podemos observar el resultado es el esperado $\rightarrow 16 = 64/4$

2. Conclusiones

Esta práctica, junto con la teoría estudiada en clase, nos ayudó bastante a entender el funcionamiento interno de lenguajes imperativos como C y algunas de las diferencias que este paradigma tiene con el funcional. Los retos encontrados durante la implementación de la práctica no fueron muy difíciles de resolver, sin embargo fue un poco tedioso codificar las soluciones debido a la necesidad de repetir múltiples veces, con mínimas variaciones, ciertos pedazos de código.