Ejercicio Semanal 3

Implementación de `MiniC`

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

Miércoles 31 de octubre de 2018

Fecha de entrega: Miércoles 7 de noviembre de 2018 a las 23:59:59.

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

Recordemos que las expresiones de `EAB` son las siguientes:

```
data Expr = V Identifier | I Int | B Bool
  | Add Expr Expr | Mul Expr Expr | Succ Expr | Pred Expr
  | And Expr Expr | Or Expr Expr | Not Expr
  | Lt Expr Expr | Gt Expr Expr | Eq Expr Expr
  | If Expr Expr Expr
  | Let Identifier Type Expr Expr
```

1 Paradigma Imperativo

A grandes rasgos podemos definir el paradigma imperativo como:

Paradigma Imperativo = Paradigma Funcional + Efectos laterales

Los efectos laterales que implementaremos son la asignación, que es un efecto de almacenamiento modelado mediante *referencias* a la memoria, y los operadores de secuencia e iteración.

1.1 Memoria

Consideremos el siguiente programa en `C`:

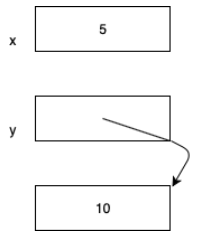
```
#include <stdio.h>
#include <stdlib.h>

int main(){
```

```
    int x = 5;
    int* y = (int*)malloc(sizeof(int));
    *y = 10;

    printf("Valor de x: %d\n", x);
    printf("Valor de la referencia y: %p\n", y);
    printf("Contenido de la referencia y: %d\n", *y);

}
```

La variable `x` de tipo `int` (entero) almacena el valor `5`, mientras que la variable `y` de tipo `int*` (apuntador de entero) almacena una referencia a una celda mutable cuyo contenido es el valor `10`.

Observemos que para asignar el valor al apuntador de `y`, primero utilizamos la función `malloc()` que recibe el tamaño del tipo a almacenar en memoria y devuelve un apuntador tipo `void`, acto seguido utilizando el operador `*` indicamos que el valor que contendrá este nuevo apuntador de `y` será `10`.

Gráficamente esto de puede representar del siguiente modo:



Modelaremos este comportamiento con los constructores:

```
...
| L Int
| Alloc Expr
| Deref Expr
| Assig Expr Expr
```

donde `L` representa una dirección de memoria, `Alloc` y `Deref` representan los operadores de alojamiento y recuperación, y `Assig` el operador de asignación.

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

```
-- Alias for memory addresses.
type Addr = Expr
-- Alias for values.
type Value = Expr

type Cell = (Addr, Value)
```

```
type Mem = [Cell]
```

Implementa las siguientes funciones:

1. (0.5 puntos) `domain`. Dada una memoria, obtiene todas sus direcciones de memoria.

   ```
   domain :: Mem -> [Int]
   ```

   Ejemplo:

   ```
   *Main> domain []
   []
   *Main> domain [(L 0, B False), (L 2, I 9)]
   [0, 2]
   *Main> domain [(L 0, I 21), (L 1, Void), (L 2, I 12)]
   [0, 1, 2]
   *Main> domain [(L 0, I 21), (L 1, Void),
                  (B False, I 12), (L 3, B True)]
   *** Exception: Corrupted memory.
   ```

2. (0.5 puntos) `newL`. Dada una memoria, genera una nueva dirección de memoria que no este contenida en esta.

   ```
   newL :: Mem -> Addr
   ```

   Ejemplo:

   ```
   *Main> newL []
   L 0
   *Main> newL [(L 0, B False), (L 2, I 9)]
   L 1
   *Main> newL [(L 0, I 21), (L 1, Void), (L 2, I 12)]
   L 3
   ```

3. (1 punto) `accessM`. Dada una dirección de memoria, devuelve el valor contenido en la celta con tal dirección, en caso de no encontrarla debe devolver `Nothing`.

   ```
   accessM :: Addr -> Mem -> Maybe Value
   ```

   Ejemplo:

   ```
   *Main> accessM (L 3) []
   Nothing
   *Main> accessM (L 1) [(L 0, B False), (L 2, I 9)]
   Nothing
   *Main> accessM (L 2) [(L 0, I 21), (L 1, Void), (L 2, I 12)]
   Just (I 12)
   *Main> accessM (L 2) [(L 0, I 21), (I 3, Void), (L 2, I 12)]
   *** Exception: Corrupted memory.
   ```

4. (1 punto) `updateM`. Dada una celda de memoria, actualiza el valor de esta misma en la memoria.

```
updateM :: Cell -> Mem -> Mem
```

Ejemplo:

```
*Main> updateM (L 3, B True) []
*** Exception: Memory address does not exist.
*Main> updateM (L 0, Succ (V "x")) [(L 0, B False), (L 2, I 9)]
*** Exception: Memory can only store values.
*Main> updateM (L 0, I 22) [(L 0, I 21), (L 1, Void), (L 2, I 12)]
[(L 0, I 22), (L 1, Void), (L 2, I 12)]
*Main> updateM (L 2, I 14) [(L 0, I 21), (I 3, Void), (L 2, I 12)]
*** Exception: Corrupted memory.
```

1.2 Ejecución Secuencial

Un mecanismo primordial en el paradigma imperativo es la ejecución de instrucciones en secuencia. La notación $e_1; e_2$ indica que se debe ejecutar e_1 y al finalizar proceder con la ejecución de e_2. En los casos de interés la ejecución de e_1 causa un efecto y no devuelve un valor, o lo que es lo mismo, el valor que devuelve no tiene interés por lo que se descarta y se representa con el valor de *void* de tipo unitario $Void$, este tipo de instrucciones se conoce como *comandos*. Los juicios de la semántica dinámica de la operación de secuencia (;) son:

$$\frac{e_1 \to e_1'}{e_1; e_2 \to e_1'; e_2}\ seq$$

$$\frac{}{void; e_2 \to e_2}\ seqv$$

Modelaremos esta operación con los constructores:

```
...
| Void
| Seq Expr Expr
```

1.3 Ciclo `While`

Para terminar con la definición de `MiniC` añadiremos el operador de iteración, en este caso el ciclo `While`. Esta instrucción recibe una guardia y un bloque de instrucciones a ejecutar. El mecanismo del ciclo `While` es sencillo, el bloque de instrucciones se ejecuta hasta que la guardia resulte falsa.

El constructor correspondiente es:

```
...
| While Expr Expr
```

Implementa las siguientes funciones:

1. (0.5 puntos) `frVars`. Extiende esta función para las nuevas expresiones.

```
frVars :: Expr -> [Identifier]
```

   Ejemplo:

```
*Main> frVars (Add (V "x") (I 5))
["x"]
*Main> frVars (Assig (L 2) (Add (I 0) (V "z")))
["z"]
```

I 2

2. (0.5 puntos) `subst`. Extiende esta función para las nuevas expresiones.

```
subst :: Expr -> Substitution -> Expr
```

   Recuerda que la definición de la sustitución es:

```
type Substitution = (Identifier, Expr)
```

   Ejemplo:

```
*Main> subst (Add (V "x") (I 5)) ("x", I 10)
Add (I 10) (I 5)
*Main> subst (Let "x" (I 1) (V "x")) ("y", Add (V "x") (I 5))
*** Exception: Could not apply the substitution.
*Main> subst (Assig (L 2) (Add (I 0) (V "z"))) ("z", B False)
(Assig (L 2) (Add (I 0) (B False)))
```

3. (3 puntos) `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 $\langle m, e\rangle \to \langle m', e'\rangle$.

```
eval1 :: (M, Expr) -> (M, Expr)
```

   Ejemplo:

```
*Main> eval1 ([(L 0, B False)], (Add (I 1) (I 2)))
([(L 0, B False)], I 3)
*Main> eval1 ([(L 0, B False)], (Let "x" (I 1) (Add (V "x") (I 2))))
([(L 0, B False)], Add (I 1) (I 2))
*Main> eval1 ([(L 0, B False)], Assig (L 0) (B True))
([(L 0, B True)], Void)
*Main> eval1 ([], While (B True) (Add (I 1) (I 1)))
([], If (B True) (Seq (Add (I 1) (I 1))
(While (B True) (Add (I 1) (I 1)))) Void)
```

4. (2 puntos) `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 $\langle m, e\rangle \rightarrow^* \langle m', e'\rangle$ y e' está bloqueado.

```
evals :: (M, Expr) -> (M, Expr)
```

Ejemplo:

```
*Main> evals ([], (Let "x" (Add (I 1) (I 2)) (Eq (V "x") (I 0))))
([], B False)
*Main> evals ([], (Add (Mul (I 2) (I 6)) (B True)))
([], Add (I 12) (B True))
*Main> evals ([], Assig (Alloc (B False)) (Add (I 1) (I 9)))
([(L 0, I 10)], Void)
```

5. (1 punto) `eval`. Devuelve la evaluación de un programa tal que `eval e = e'` si y solo si $\langle \emptyset, e\rangle \rightarrow^* \langle m', e'\rangle$ y e' es un valor. En caso de que e' no sea un valor deberá mostrar un mensaje de error particular al operador que lo causó.

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

Ejemplo:

```
*Main> eval (Add (Mul (I 2) (I 6)) (B True))
*** Exception: [Add] Expects two Nat.
*Main> eval (Or (Eq (Add(I0 ) ( I0) )( I0) ) (Eq (I 1) (I 1 0)))
B True
*Main> eval (While (B True) Void)
```

6. (1 punto) Brinda una expresión en `MiniC` que represente alguno de los programas que utilice el ciclo *while* descritos en la nota de clase 10 (páginas 10 - 11), ejecútala y verifica que funciona correctamente.

   Ejemplo:

```
whileTrue = While (B True) Void

fact n =
Let "x" (Alloc n) (
Let "y" (Alloc (I 1)) (
    Seq
    (
        While (Gt (Deref (V "x")) (I 0))
        (Seq
        (Assig (V "y") (Mul (Deref (V "x")) (Deref (V "y"))))
        (Assig (V "x") (Add (Deref (V "x")) (I (-1))))
        )
```

```
        )
        (
            Deref (V "y")
        )
    )
)
```

¡Suerte!