Práctica 5

Implementación de `MiniC`

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

Lunes 28 de octubre de 2019

Fecha de entrega: Miércoles 6 de noviembre de 2019 a las 23:59:59.

`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.

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

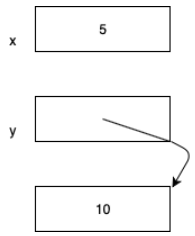
Consideremos el siguiente programa en `C`:

```
#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 Address = Int
-- Alias for values.
type Value = Expr

type Cell = (Address, Value)

type Memory = [Cell]
```

Implementa las siguientes funciones en un módulo llamado `Memory`:

1. (0.5 puntos) `newAddress`. Dada una memoria, genera una nueva dirección de memoria que no este contenida en esta.

   ```
   newAddress :: Memory -> Expr
   ```

   Ejemplo:

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

2. (1 punto) `access`. Dada una dirección de memoria, devuelve el valor contenido en la celta con tal dirección, en caso de no encontrarla debe devolver `Nothing`.

```
access :: Address -> Memory -> Maybe Value
```

Ejemplo:

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

3. (1 punto) `update`. Dada una celda de memoria, actualiza el valor de esta misma en la memoria. En caso de no existir debe devolver `Nothing`.

```
update :: Cell -> Memory -> Maybe Memory
```

Ejemplo:

```
*Main> update (3, B True) []
Nothing
*Main> update (0, Succ (V "x")) [(0, B False)
    , (2, I 9)]
*** Exception: Memory can only store values.
*Main> update (0, Fn "x" (V "x")) [(0, I 21),
     (1, Void), (2, I 12)]
[(0, Fn "x" (V "x")), (1, Void), (2, I 12)]
*Main> update (2, I 14) [(0, I 21), (2, Void)
    , (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 o extiende 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"]
   ```

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))
Let "x1" (I 1) (V "x1")
*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 :: (Memory, Expr) -> (Memory, Expr)
```

Ejemplo:

```
*Main> eval1 ([(0, B False)], (Add (I 1) (I
    2)))
([(0, B False)], I 3)
*Main> eval1 ([(0, B False)], (Let "x" (I 1)
    (Add (V "x") (I 2))))
([(0, B False)], Add (I 1) (I 2))
*Main> eval1 ([(0, B False)], Assig (L 0) (B
    True))
([(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 $\langle m, e\rangle \to^* \langle m', e'\rangle$ y e' está bloqueado.

```
evals :: (Memory, Expr) -> (Memory, 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)))
([(0, I 10)], Void)
```

5. (1,5 puntos) `evale`. Extiende esta función para que dada una expresión, devuelva la evaluación de un programa tal que `evale e = e'` syss $e \to^* e'$ 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.

   ```
   evale :: Expr -> Expr
   ```

   Ejemplo:

   ```
   *Main> evale (Add (Mul (I 2) (I 6)) (B True))
   *** Exception: [Add] Expects two Integer.
   *Main> evale (Or (Eq (Add (I 0) (I 0)) (I 0))
       (Eq (I 1) (I 10)))
   B True
   *Main> evale (While (B True) Void)
   ```

6. (1 punto) Agrega al directorio `demo` la implementación de alguno de los programas (distinto de los siguientes) que utilice el ciclo *while* descritos en la nota de clase 10 (páginas 7 - 8) en `MiniC`, ejecútala y verifica que funciona correctamente.

   Ejemplo:

   ```
   //Infinite loop
   while (true) do
       void

   //Example of a program of iterative factorial
      function.
   let fac := fn n => let x := alloc n in
                        let y := alloc 1 in
                          (while (!x > 0) do
                            y ::= !x * !y;
                            x ::= pred !x
                           end);
                          !y
                        end
                      end
   ```

```
in
  fac $ 5
end :: Integer
```

¡Suerte!