Principles of Programming Languages (Sheet #2)

Marius Gavrilescu

1. ```
   val x = new();;
   val y = new();;
   x := 1;;
   y := 2;;
   !x;;
   ```

   produces 1 instead of 2, so we know x and y refer to different addresses in memory.

2. No, we cannot take a reference to a variable in FunMem (all we can do is create a reference to a brand new memory location). The memory and the environment are different namespaces in the interpreter, and variables are part of the environment (not the memory). While a variable's value can be a reference to a memory location, a variable's value cannot be a reference to another variable.

   Using ref cells we can do:

   ```
   val last = new();;

   rec cons(lst,e) =
     let val cdr = new() in
     cdr := lst; e : cdr
   ;;

   rec hd(lst) = head(lst);;
   rec tl(lst) = !tail(lst);;

   val lst = 5 : last;;
   val lst = cons(lst, 4);;
   val lst = cons(lst, 3);;
   val lst = cons(lst, 2);;

   hd(lst);;
   hd(tl(lst));;
   hd(tl(tl(lst)));;

   last := lst;;

   hd(lst);;
   hd(tl(lst));;
   hd(tl(tl(lst)));;
   hd(tl(tl(tl(lst))));;
   hd(tl(tl(tl(tl(lst)))));;
   hd(tl(tl(tl(tl(tl(lst))))));;
   ```
   ```

3. We can evaluate "repeat" expressions like this:

```
eval (Repeat e2 e1) env = eval e2 env $> (\v2 -> u)
  where
    u = eval e1 env $> (\v1 ->
      case v1 of
   BoolVal False -> eval e2 env $> (\v2 -> u)
BoolVal True  -> result Nil
_ -> error "boolean required in repeat loop")
```

4. The new primitive is:

```
primitive "put" (\ [Addr a, Value v] -> put a v),
```

6. `type M a = Mem -> (a, Mem, String)`

```
result :: a -> M a
result x mem = (x, mem, "")

($>) :: M a -> (a -> M b) -> M b
(xm $> f) mem =
  let (x, mem1, out1) = xm mem in
  let (y, mem2, out2) = (f $! x) mem1 in
  (y, mem2, out1 ++ out2)

new :: M Location
new mem = let (a, mem') = fresh mem in (a, mem', "")

get :: Location -> M Value
get a mem = (contents mem a, mem, "")

put :: Location -> Value -> M ()
put a v mem = ((), update mem a v, "")

output :: String -> M ()
output out mem = ((), mem, out)
```

7. Looking at (my version of) eval:

```
eval (Apply f es) env =
  eval f env $> (\fv ->
    let ev x = Thunk x env in
    let xargs = map ev es in
    apply fv xargs)
```

   we see that f is evaluated first, and then the paramters are turned into thunks left-to-right (= evaluated left-to-right in FunMem).

   We can change this to do the opposite:

```
eval (Apply f es) env =
  let ev x = eval x env in
  let xargs = reverse (map ev (reverse es)) in
  eval f env $> (\fv -> apply fv xargs)
```

5. We have:

```
(result v) $> f
=
(\ mem -> (v, mem)) $> f
=
(\ mem -> let (x, mem') = (\ mem -> (v, mem)) mem in (f $! x) mem')
=
(\ mem -> let (x, mem') = (v, mem) in (f $! x) mem')
=
(\ mem -> f v mem)
=
f v

m $> result
=
(\ mem -> let (x, mem') = m mem in (result $! x) mem')
=
(\ mem -> let (x, mem') = m mem in (x, mem'))
=
(\ mem -> m mem)
=
m

(m $> f) $> g
=
(\ mem -> let (x, mem') = (m $> f) mem in (g $! x) mem')
=
(\ mem ->
  let (x, mem') =
    let (x', mem'') = m mem in
    (f $! x') mem''
  in
  (g $! x) mem')
=
(\ mem ->
  let (x', mem'') = m mem in
  let (x, mem') = (f x') mem'' in
  (g $! x) mem')
=
(\ mem -> let (x', mem'') = m mem in (f x' $> g) mem'')
=
(\ mem -> let (x', mem'') = m mem in (\x -> (f x' $> g)) x mem'')
=
m $> (\x -> (f x $> g))
```

8. ```
   eval (Seq e1 (Seq e2 e3)) env
   =
   eval e1 env $> (\ v -> eval (Seq e2 e3) env)
   =
   eval e1 env $> (\ v -> eval e2 env $> (\w -> eval e3 env))
   = { Monad associativity }
   (eval e1 env $> (\ v -> eval e2 env)) $> (\w -> eval e3 env)
   =
   eval (Seq e1 e2) env $> (\w -> eval e3 env)
   =
   eval (Seq (Seq e1 e2) e3) env


   eval (Seq (Variable "nil") e1) env
   =
   eval (Variable "nil") env $> (\ v -> eval e1 env)
   =
   result Nil $> (\ v -> eval e1 env)
   = { Monad left identity }
   (\ v -> eval e1 env) Nil
   =
   eval e1 env


   eval (Seq e1 e2) env
   =
   eval e1 env $> (\ v -> eval e2 env)
   =
   eval e1 env $> (\v -> (\ env' -> eval e2 env') env)
   = { Monad left identity }
   eval e1 env $> (\ v -> result env $> (\ env' -> eval e2 env'))
   =
   eval e1 env
   $> (\ v -> (\ v -> result env) v $> (\ env' -> eval e2 env')
   = { Monad associativity }
   eval e1 env
   $> (\ v -> result env)
   $> (\ env' -> eval e2 env')
   = { "x" is unused }
   eval e1 env
   $> (\ v -> result (define env "x" (Const v))
   $> (\ env' -> eval e2 env')
   =
   elab (Val "x" e1) env $> (\ env' -> eval e2 env')
   =
   eval (Let (Val "x" e1) e2) env
   ```

   `e1; nil` is not equivalent to `e1` because the former discards the value of `e1`.

9. ```
   mapm f [] = []
   mapm f (x:xs) = f x $> (\ v -> mapm f xs $> (\ vs -> result (v : vs)))
   ```