Mutation in the Interpreter

We will discuss in this section the challenges involved in introducing mutation into an interpreter. This will lead us to the need for a second storage unit in addition to the environment, with a distinctly different behavior. We refer to this second unit as the store.

The problem with our current Interpreter

Let us consider the problem with using our current interpreter in a language that we want to introduce mutation. Our interpreter so far operates as follows:

- There is an environment in which bindings are looked up.
- The interpreter evaluates an expression in a given environment.
- The interpreter returns a value as the result of the evaluation.
- The environment extends in `let` expressions and function calls.
- The environment gets stored in a function closure, to be retrieved at the function call later.
- Expressions with multiple subparts typically recursively call themselves on the subparts, with the same environment.

We will see that the last two properties, which are essential parts of the interpreter's behavior, will force us to make changes for an interpreter that needs to accomodate mutation.

Let us start by considering the standard addition, which looked something like this in the interpreter, ignoring type-checking safeguards:

```
[(plus? e) (+ (interp env (plus-e1 e))
              (interp env (plus-e2 e)))]
```

Here `env` is the environment in which `e` was to be evaluated. The key emphasis here is that it is this same environment that is used to evaluate both terms of the expression.

But in the presence of mutation, we need to accept the possibility that something that we do in the first summand might affect the second summand. For instance in the following Racket code:

```
(let ([a 3])
  (+ (begin (set! a 2) 4)
     a))
```

The answer should be `6`, not `7`. But in the above interpretation there is no room for that. The interpretation of the first summand is simply returning a value, and has no way to communicate the change that has occured to `a`. This is a major problem.

> In order to accomodate mutation, the interpreter needs to be able to return more than a value, in order to communicate possible changes to the next step in time. The standard behavior of the environment in an interpreter does not incorporate the time factor.

So what should the interpreter return, in addition to a value? One possible answer would be that it needs to return a new environment, but this would be wrong. A good reason for that is the way closures work. For instance consider this example:

```
(define b 2)
(define f (lambda () b))
(f)
(set! b 5)
(f)
```

There is an environment that gets stored along with the definition of `f`, to form the closure. In any subsequent calls to `f`, that environment is accessed and `f` is evaluated in that environment. We have no way to change that environment, and so we have no way to make the two calls to `f` produce different results, as they should.

> Closures store an environment in which the function is to be evaluated. The presence of mutation means that we need to somehow be able to change the values of the variables bound in that environment, without actually changing the environment.

This leads us to a key observation: The environment cannot actually hold values. It must hold "locations". And those locations must then link to values. It is that linking that may change over time as assignments and mutation take place.

A good analog is the difference between the stack and the heap. The environment is like the stack, typically holding pointers to memory locations on which the data is actually stored. Variables are bound to these locations, and changes in those bindings relate to adding and removing calls from the stack, which is loosely the analog to let-expressions extending the environment. But we also need something like the **heap**, which relates memory locations to data stored at those locations, and that data may change without the locations changing.

We therefore split the storage of the information needed by the interpreter in two parts:

- The **environment** binds variables to **locations**. It is part of the input to the interpreter, and it behaves in a recursive decent way, extending on let statements and function calls, and being stored in closures.
- The **store** relates locations to values. It is returned from each interpreter call, along with the value produced, and is provided as input to the following call to propagate value changes due to mutation. This is called **storage-passing style**.
- There needs to be a mechanism for creating new "locations", which more or less amounts to a `malloc` call for memory allocation. We will "emulate" that process by having locations represented by integers, and a new location being produced by simply increasing an index.

A minimal interpreter with structure mutation

```
; Locations are integers.
; (new-loc) returns a new "location".
; (bind s loc env) adds a binding of a symbol to a location and
; extends the environment.
; (lookup s env) looks a symbol up in an environment, returns a
; location.
; (store loc v sto) extends the store by associating the value
; to the location.
; (fetch loc sto) looks up a location in the store and returns
; the value stored there.
;
; Language expressions:
(struct var (s) #:transparent)
(struct num (n) #:transparent)
(struct arith (op e1 e2) #:transparent)
(struct fun (fname arg body) #:transparent)
(struct call (e1 e2) #:transparent)
(struct letC (s e1 e2) #:transparent)
(struct ref (e) #:transparent)
(struct assign (e1 e2) #:transparent)
(struct deref (e) #:transparent)
(struct seq (e1 e2) #:transparent)
;
; Values:
(struct numV (n) #:transparent)
(struct closV (f env) #:transparent)
(struct refV (loc) #:transparent)
; Results are a value plus a store:
(struct res (v sto) #:transparent)
; interp: env sto e  -> result
(define (interp env sto e)
  (cond
    [(num? e) (res (numV (num-n e))
                   sto)]
  ... ; will look at them one at a time
  ))
```

The interpreter now takes as input an environment and a store and an expression, and returns a pair of a value and an updated store. Some cases don't really change the store, they just propagate the provided store. Others need to change it.

We start with the var case. It must use lookup to get the location where the value is stored, then use fetch to grab the value stored in that location.

```
    [(var? e)
     (res (fetch (lookup (var-s e) env) sto)
          sto)]
```

Arithmetic operations must be careful about the order of evaluation: The first expression must be evaluated, returning a value and an updated store. Then *this updated store* must be used when evaluating the second expression. Finally, we must return the store returned by that second expression.

```
    [(arith? e)
```

```
 (let* ([r1 (interp env sto (arith-e1 e))]
        [r2 (interp env (res-sto r1) (arith-e2 e))])
   (if (and (numV? (res-v r1))
            (numV? (res-v r2)))
       (res (numV ((arith-op e) (numV-n (res-v r1))
                                (numV-n (res-v r2))))
            (res-sto r2))
       (error "interp: adding non-numbers"))))]
```

Notice how much more complicated each case becomes out of the need to properly maintain the time-effect of store-passing style.

We'll now look at the cases that are related to mutation. seq is similar to arith: e1 is interpreter only so we can get an updated store.

```
[(seq? e)
 (let ([r1 (interp env sto (seq-e1 e))])
   (interp env (res-sto r1) (seq-e2 e)))]
```

ref needs to create a new location, then store the value from the expression.

```
[(ref? e)
 (let ([loc (new-loc)]
       [r (interp env sto (ref-e e))])
   (res (ref loc)
        (store loc (res-v r) (res-sto r))))]
```

unref is straightforward. It must ensure that its expression evaluates to a refV:

```
[(unref? e)
 (let ([r (interp env sto (unref-e e))])
   (if (refV? (res-v r))
       (res (fetch (refV-loc (res-v r))
                   (res-sto r))
            (res-sto r)))
   (error "unreferencing from non-reference"))]
```

Finally, we have to implement assign. Since we don't have "unit"s in our language we will simply return the value. But it is important to update the store:

```
[(assign? e)
 (let* ([r1 (interp env sto (assign-e1 e))]
        [r2 (interp env (res-sto r1) (assign-e2 e))])
   (if (refV? (res-v r1))
       (res (res-v r2)
            (store (refV-loc (res-v r1))
                             (res-v r2)
                             (res-sto r2)))
       (error "assigning to non-reference"))))]
```

Next, we have to implement function definitions and function calls, as well as let statement. We start with the let. letC needs to bind the symbol to a location that holds a value. So it needs to create a new location and update both the environment and the store before evaluating the second expression. So this is how that will look (pay attention to the updated store):

```
[(letC? e)
 (let* ([loc (new-loc)]
        [r1 (interp env sto (letC-e1 e))])
   (interp (bind (letC-s e) loc env)
           (store loc (res-v r1) (res-sto r1))
           (letC-e2 e)))]
```

Function definition is straightforward:

```
[(fun? e)
 (res (closV e env)
      sto)]
```

Function calls are trickier but similar to letC. We need to: Evaluate the function location to a value, evaluate the parameter location to a value, check that the function value is indeed a closure, create a new location to hold the binding of the parameter, get the environment out of the closure and extend it by this new location, extend the appropriate scope with this new location related to the parameter's value, then evaluate the body in the appropriate environment and scope. Phew. Ok here's how that looks (ignoring for the sake of simplicity the question of the recursive name for the function):

```
[(call? e)
 (let* ([loc (new-loc)]
        [rf (interp env sto (call-e1 e))]
        [rv (interp env (res-sto rf) (call-e2 e))]
        [cl (res-v rf)])
   (if (closV? cl)
       (interp (bind (fun-arg (closV-f cl))
                     loc
                     (closV-env cl))
               (store loc
                      (res-v rv)
                      (res-sto rv))
               (fun-body (closV-f cl)))
       (error "Trying to call non-function"))))]
```

A very interesting question arises if the value to be passed is already a refV, with its own location. Should we reuse that location, instead of creating our own new location? What does this mean about the semantics of the function call?