Interpreting objects and classes

Introduction

Building on our work with an interpreter that allows mutation, we will now further discuss how to implement objects and classes in it. Let us think of the main ingredients:

- There are two new values: classes and objects
- A class contains a constructor/initializer, a list of symbols for the field names and methods. Classes don't need their own "names" as they will be regular values.
- A class can have an optional superclass, which must be an expression that evaluates to a class. We can thus have inheritance. If the "superclass" location evaluates to something other than a class, then we simply assume our class doesn't have a superclass.
- Objects are created by calling the constructor/initializer. This is done by using a `new` construct. For simplicity, the constructors take no arguments. It is up to the constructors to decide on initial values.
- When the constructor is called, a new object is created with all the fields assigned memory locations but without any values. The constructor is then called with that object as `self`. If there is a superclass, its constructor is called first.
- You cannot add new fields to an object, nor delete fields.
- All superclasses of the object have access to its fields. This is technically not ideal, but it will make our implementation simpler.
- The result of a call to `new` is the object that was created, regardless of what the constructor returns. This is a design choice to some extent.
- We can call object methods by using dynamic dispatch.
- Methods are stored as functions, but their call/dispatch semantics is slightly different.
- We assume that functions take only one argument.
- The object's fields are accessed through get/set methods, analogous to `o.x` and `o.x = v`.
- The keyword `self` is used to refer to the object from within a method call. We have to decide how to make that happen:
  - One approach is to simply add the symbol `'self` in the environment at appropriate times. There are two problems with this. The one is that one could shadow the object by directly using `'self` as a variable. The other more important problem is that the environment binds symbols to locations. This would mean that every time we try to implement method dispatch, we have the object value and must create a new memory location for it for the purposes of the call.
  - Another approach is to add another parameter to the `interp` call, is the "current object". Then we can use a keyword `self` that simply returns that object. This is the approach we take. If there is no "current object" then the object parameter can contain `#f` or any other 'meaningless' statement.

Here's how the main structs for something like this might look like:

```
;; New language construts
(struct classC (super init fnames mnames methods))
(struct newC (class))
(struct self ())
(struct disp (obj s arg))  ;; method call/dispatch. arg is an expression
(struct getC (obj s))     ;; returns the value in field s
(struct setC (obj s e))   ;; sets the value in field s to the result of interp e
;; New values
(struct classV (super init fnames mnames methods))
(struct objV (class fields locs) #:transparent)
```

Let us consider implementation. First, because we will encounter the situation of having a racket list of expressions that need to be evaluated, each passing its store to the next, we should create a helper function to do that:

```
(define (interp-list o env sto lst)
  (if (null? lst)
      (res null sto)
      (let* ([r (interp o env sto (car lst))]
             [r-rest (interp-list o env (res-sto r) (cdr lst))])
        (res (cons (res-v r)
                   (res-v r-rest))
             (res-sto r-rest)))))
```

Classes

Evaluating a classC is simple, but there's plenty of information stored there:

- A classC will evaluate to a classV once it makes sure all components are in place.
- We must evaluate super to a value.
- fnames must be a list of symbols with no duplicates (but we won't check for duplicates). Only methods of the class can access the field names of the class. For example methods of a subclass can't directly access a field defined in the superclass. Implementing this part isn't easy.
- init should evaluate to a closure.
- mnames is a list of method names. It should have the same length as methods.
- methods should be a list of expressions, where the expressions evaluate to closures.
- As each of these is evaluated, we must maintain the store and propagate it.

```
;; (classC super init fnames mnames methods)
    [(classC? e)
     (let* ([r-super (interp o env sto (classC-super e))]
            [super (res-v r-super)]
            [r-init (interp o env (res-sto r-super) (classC-init e))]
            [init (res-v r-init)]
            [r-methods (interp-list o env (res-sto r-init) (classC-methods e))]
            [methods (res-v r-methods)]
            [fnames (classC-fnames e)]
```

```
          [mnames (classC-mnames e)])
      (cond [(not (andmap symbol? fnames))
             (error "field names must be symbols")]
            [(not (andmap symbol? mnames))
             (error "method names must be symbols")]
            [(not (closV? init))
             (error "initializer must be a function")]
            [(not (andmap closV? methods))
             (error "class methods must be functions")]
            [else
             (res (classV super init fnames mnames methods)
                  (res-sto r-methods))]))]
```

New objects

In order to implement new, we must do the following:

- We must get hold of the chain of superclasses of the object.
- We must extract the field symbols from all superclasses, and form a single list of symbols for the object.
- We then build a basic object. We must allocate memory locations for all fields.
- We call all initializers in order, starting from the super-most one. The last initializer to run is the one corresponding to the object's class.
- We return the object.

Let us start with some helper methods. Here's a method that forms the superclass list from a base class:

```
(define (get-class-list cls)
  (letrec ([extend
            (lambda (classes)
              (let ([next (classV-super (car classes))])
                (if (classV? next)
                    (extend (cons next classes))
                    classes))))])
    (extend (list cls))))
```

Then a method that is given an class for an object, and forms a list of the unique symbols from the object's superclasses. In theory we could do this once and store it in classV. We also need a method uniq to remove duplicates.

```
(define (uniq xs seen)
  (cond [(null? xs) seen]
        [(memq? (car xs) seen)
         (uniq (cdr xs) seen)]
        [else (uniq (cdr xs) (cons (car xs) seen))]))

(define (get-symbols cls)
  (let* ([classes (get-class-list cls)]
         [symbols (flatten (map classV-fnames classes))])
    (uniq symbols null)))
```

```
(define (get-inits cls)
  (map classV-init (get-class-list cls)))
```

We'll need a function to call all initializers on an object, and return the updated store. Each initializer needs to be called in its own environment, stored in its closure. There are no arguments so we simply need to reach into the function body and evaluate that.

```
(define (call-inits obj sto inits)
  (if (null? inits)
      sto
      (let* ([init (car inits)]
             [cl-env (closV-env init)]
             [body (fun-body (closV-fun init))]
             [new-sto (res-sto (interp obj cl-env sto body))])
        (call-inits obj new-sto (cdr inits)))))
```

Now we are ready to implement new:

- It must look up the expression at the class position, and make sure it is a class.
- We need to then create a new object value, and create locations for all the fields.
- We then call the class initializers with the correct self object.
- Finally we return the object.

```
;; (newC class)
   [(newC? e)
    (let* ([r-cls (interp o env sto (newC-class e))]
           [cls (res-v r-cls)])
      (if (classV? cls)
          (let* ([fields (get-symbols cls)]
                 [locs (map (lambda (t) (new-loc)) fields)]
                 [obj (objV cls fields locs)]
                 [rinit (call-inits obj
                                    (res-sto r-cls)
                                    (get-inits cls))])
            (res obj (res-sto rinit)))
          (error "new needs a class")))]
```

Method Dispatch

Next up we need to implement method dispatch. We need to:

- Evaluate in the current object/environment/store the value that is in the object place.
- Evaluate the argument to a value in the current object/environment and the updated store.
- Search through the superclass hierarchy for the implementation of a method with symbol s.

- Perform the call with the correct object/enviroment.

```
;; (disp obj s arg)
    [(disp? e)
     (let* ([robj (interp o env sto (disp-obj e))]
        [obj (res-v robj)]
        [rv (interp o env (res-sto robj) (disp-arg e))]
        [v (res-v rv)])
       (if (objV? obj)
           (call-method (find-method (disp-s e) obj) obj (res-sto rv) v)
           (error "cannot dispatch on non-object"))))]
```

We need to implement call-method and find-method. We start with call-method. It is given a function value, an object to use as self, a store, and an argument value. It does not need an environment as it will use the function closure's environment. It *does* need to create a new memory location to store the argument.

We would normally check that the method we are given is actually a closure. But we have ensured that when we created the classes. So we would never dispatch on a non-method.

```
(define (call-method fv obj sto argv)
  (let ([f (closV-f fv)]
        [loc (new-loc)])
    (interp obj
            (bind (fun-arg f) loc (closV-env fv))
            (store loc argv sto)
            (fun-body f))))
```

Now find-method, the main method that implements the dynamic dispatch. It is given a symbol and an object, and it must locate the appropriate method to call by going up the chain. It uses a helper method, that takes a list of names and methods and if it finds the symbol it returns the corresponding method.

```
(define (search-in-list s mnames methods)
  (cond [(null? mnames) #f]
        [(eq? s (car mnames)) (car methods)]
        [else (search-in-list s (cdr mnames) (cdr methods))]))

(define (check-in-class s cls)
  (let* ([method
          (search-in-list s
                          (classV-mnames cls)
                          (classV-methods cls))])
    (cond [method method]
          [(classV? (classV-super cls))
           (check-in-class s (classV-super cls))]
          [else (error "Unknown method")])))

(define (find-method s obj)
  (check-in-class s (objV-class obj)))
```

Getters and Setters

Lastly, we must implement getC/setC functionality. We could possibly generalize it in terms of the concept of "l-values", but we'll keep it simple.

Interpreting the getter, which is given an object expression and symbol must:

- Evaluate the object expression to a value.
- Ensure the value is an object.
- Check that the object has the corresponding field and find the location it points to.
- Fetch the value at that location.

We will reuse search-in-list from earlier. We will also use a helper function get-field-loc that gets the location of a field from the object.

```
(define (get-field-loc s obj)
  (let* ([fields (objV-fields obj)]
         [locs (objV-locs obj)]
         [loc (search-in-list s fields locs)])
    (or loc (error "Object does not contain field"))))
```

Now for getC:

```
;; (getC obj s)
[(getC? e)
 (let* ([robj (interp o env sto (getC-obj e))]
        [obj (res-v robj)])
   (if (objV? obj)
          (res (fetch (get-field-loc (getC-s e) obj)
                      (res-sto robj))
               (res-sto robj))
       (error "Cannot get field of non-object"))))]
```

Similarly for setC, with extra steps for evaluating the value.

```
;; (setC obj s e)
[(setC? e)
 (let* ([robj (interp o env sto (setC-obj e))]
        [obj (res-v robj)]
        [rv (interp o env (res-sto robj) (setC-e e))]
        [v (res-v rv)])
   (if (objV? obj)
       (let ([loc (get-field-loc (setC-s e) obj)])
         (res v (store loc v (res-sto rv))))
       (error "Cannot set field of non-object"))))]
```

This essentially completes our basic implementation of classes and objects.

Here is an example of a program written in this language. It creates a point class with x and y fields, then a subclass for 3-dimensional points with an extra z field. Also a "length squared" method for computing the squared distance from the origin.

```
(define prog1
  (letC 'Point
        (classC
         (voidC)
         (fun #f #f
              (seq (setC (self) 'x (num 0))
                   (setC (self) 'y (num 0))))
         (list 'x 'y)
         (list 'getX 'getY 'setX 'setY 'lensq)
         (list (fun #f #f (getC (self) 'x))
               (fun #f #f (getC (self) 'y))
               (fun #f 'newx (setC (self) 'x (var 'newx)))
               (fun #f 'newy (setC (self) 'y (var 'newy)))
               (fun #f #f
                    (arith +
                           (arith *
                                  (disp (self) 'getX (voidC))
                                  (disp (self) 'getX (voidC)))
                           (arith *
                                  (disp (self) 'getY (voidC))
                                  (disp (self) 'getY (voidC)))))))
        (letC 'Point3
              (classC
               (var 'Point)
               (fun #f #f
                    (setC (self) 'z (num 0)))
               (list 'z)
               (list 'getZ 'setZ 'lensq)
               (list (fun #f #f (getC (self) 'z))
                     (fun #f 'newz (setC (self) 'z (var 'newz)))
                     (fun #f #f
                          (arith +
                                 (arith +
                                        (arith *
                                               (disp (self) 'getX (voidC))
                                               (disp (self) 'getX (voidC)))
                                        (arith *
                                               (disp (self) 'getY (voidC))
                                               (disp (self) 'getY (voidC))))
                                 (arith *
                                        (disp (self) 'getZ (voidC))
                                        (disp (self) 'getZ (voidC)))))))
              (letC 'p (newC (var 'Point3))
                    (seq (disp (var 'p) 'setX (num 2))
                         (seq (disp (var 'p) 'setZ (num 4))
                              (disp (var 'p) 'lensq (voidC)))))))))
```