CS403: Programming Languages

Notes on Evaluating Object Orientation

Object orientation

An object oriented language provides for encapsulation, inheritance, and polymorphism. First, we focus on encapsulation.

Encapsulation

Recall that in Scheme, one can encapsulate by returning a dispatch function, which essentially serves as an intermediary between the user of the "object" and the local environment where the instance variables and methods are stored. The dispatch function translates messages to function calls and variable accesses. With Scam, one dispenses with the dispatch function and lets the user of the object deal with the local environment directly. It does so by revising the *evalFuncCall* function to insert a variable named *this* whose value is the local environment. Note that the local environment is the extended environment under which the body of the function is evaluated:

```
function evalFuncCall(t,env)
    {
    var closure = eval(getFuncCallName(t),env);
    var args = getFuncCallArgs(t);
    var params = getClosureParams(closure);
    var body = getClosureBody(closure);
    var senv = getClosureEnvironment(closure);
    var eargs = evalArgs(args,env);
    var xenv = EnvExtend(senv,params,eargs);

    //insert a variable that points to xenv
    EnvInsert(xenv, new Lexeme(ID,"this"), xenv);

    return eval(body,xenv);
    }
```

Now functions can return the local environment. Here is a simple *node* class written in the same style as our pseudocode:

```
function node(value,next)
    {
    this;
    }

var n = node(5, null);
```

The variable *n* is bound to the return value of the function, which is the local environment in which *value* and *next* are bound. This environment is also known as an *object*. Functions that return this are known as constructors as well as classes.

What is needed now is a way to access the local variables contained in an object. Assume the *dot* operator performs this function, as in:

```
println(n . value);
```

The left hand side of the dot operator should evaluate to an object, while the right hand side should be a variable name. Here is a version of *evalDot* which treats the expression as a value (rather than as a place to assign):

```
//precondition: lhs evals to object, rhs is a variable
function evalDot(t, env)
    {
    var object = eval(getLHS(t), env);
    return eval(getRHS(t), object); // objects == environments!
    }
```

One has to update the assignment operator to handle the case when the dot operator expression is used as a location:

```
function evalAssignment(t, env)
    {
    var result = eval(getRHS(t),env);
    if (getLHS(t).type == VARIABLE)
        EnvUpdate(env,getLHS(t),result);
    else if (getLHS(t).type == DOT)
        {
        var object = eval(getLHS(getLHS(t)),env);
        EnvUpdate(object,getLHS(getRHS(t)),result);
        }
    else
        fatal("bad assignment!");

    return result;
    }
```

With these changes, your language has added encapsulation to its feature set.

Adding inheritance

Inheritance can be handled using functions written in your language by manipulating the static scopes of your objects and their methods. Let:

$$A \rightarrow B \rightarrow C \rightarrow D$$

be an inheritance hierarchy with A being the most sub-class (the *child* class) and D being the most super class. Assume, in addition to the constructors A through D, we have the built-in functions *getEnclosingScope* and *setEnclosingScope* with semantics suggested by their names. Let's also assume we have four objects a, b, c, and d which are objects created by constructors A, B, C, and D, respectively.

To implement the inheritance hierarchy, we need to perform the following actions:

1. set the enclosing scope of d to the enclosing scope of a
2. set the enclosing scope of a to b, b to c, and c to d
3. set the definition pointers of the function objects/closures contained in b, c, and d to a

What step 2 does is to allow us to reference functions and variables defined in b, c, and d from a. Suppose we wish to reference a variable g via a, as in:

```
a . g;
```

If a does not contain a binding for g, then its enclosing scope is checked. Since the enclosing scope of a has been set to b, b is then searched, then c, and then d. By virtue of step 1, the original enclosing scope of a is searched if a binding for g is not found in any of the objects.

Step 3 is required so that variables referenced in functions defined in the super classes always refer to definitions in the most base class. For example, suppose all four objects, a, b, c, and d, define a function named f. Suppose further that d defines a function g that calls a function f. When the hierarchy is completed, consider a call to g via a:

```
h = a . g;
h();
```

Since g calls f, with many candidates for f, whose version of f should g call? In most modern languages, a?s version is the desired version. By virtue of step three, the definition environment of g has been switched from d to a. Thus, the new environment that comes into being from calling g will point to a. When the variable f needs to be resolved, a will be the first place a binding for f will be found. Without this step, d?s version of f would be found instead when the body of g is executed.

Generalizing the construction of inheritance hierarchies

Scam has a function named *mixin* that is found in *inherit.lib*. The *mixin* function can be used to create the hierarchy in the previous section by making the following call:

```
a = mixin(list(A(),B(),C(),D()));
```

where the list function collects the given items in a list. The calls to A, B, C, and D, create the objects referred to by a, b, c, and d in the previous section.

The *mixin* function might something like this in our pseudocode:

```
function mixin(objects)
    {
    var child = car(objects);
    setEnclosingScope(last(objects),child);          ;step 1
    chainScopes(child,cdr(objects));                 ;step 2
    setDefinitionScopes(cdr(objects),child);         ;step 3
    child;
    }
```

Note that the *mixin* function is written in your language, not the host language, and that it uses the built-in functions *car*, *cdr*, and *setEnclosingScope* (which are implemented in your host language).

The first expression after the definition of *child* implements step 1 of the previous section. The *chainScopes* function implements step 2 and looks something like:

```
function chainScopes(first,rest)
    {
    while (rest != NULL)
        {
        setEclosingScope(first,car(rest));
        chainScopes(car(first),cdr(rest));
        rest = cdr(rest);
        }
    }
```

It is also written in your language. Finally, the *setDefinitionScopes* function implements step 3:

```
function setDefinitionScopes(scopes,child)
    {
    while (scopes != NULL)
        {
        var current = car(scopes);
        var vars = getLocalVariables(current);
        var vals = getLocalValues(current);
        while (valid? vars)
            {
            if (closure?(car(vars)))
                {
                setDefiningScope(car(vars),child);
                }
            vars  = cdr(vars);
            vals  = cdr(vals);
            }
        scopes = cdr(scopes);
        }
    }
```

where the *closure?* predicate determines whether or not a variable value is a function object/closure and where *setDefiningScope* resets the definition scope pointer of a single function object/closure. The actual implementation of *setDefiningScope* is dependent on the structure of the object/closure. The functions *car*, *cdr*, *setDefiningScope*, *getLocalVariables*, and *getLocalValues* are built-in, implemented in your host language.

Generating implicit hierarchies

The *mixin* function requires that the inheritance hierarchy be explicitly passed. One can provide for an implicit generation of the hierarchy by requiring object to define a *parent* pointer. For example, the definition of A might be look like:

```
(define (A)
    (define parent (B))
    ...
    this
    )
```

The definition of B would likewise contain a parent pointer that points to an object constructed by C and so on. With the parent pointers in place, one simply needs to pass an A object to a function that extracts the inheritance hierarchy by following the parent pointers. This function then passes the resulting chain of objects to *mixin*. Scam calls this function *new* and it could be implemented as:

```
function new(child)
    {
    var chain = followParentPointers(child);
    mixin(chain);
    child;
    }
```

The *followParentPointers* function might look like:

```
function followParentPointers(current)
    {
    if (current == NULL)
        {
        "ok";
        }
    else
        {
        cons(current,followParentPointers(current . parent));
        }
    }
```

where *cons* is a built-in function. Note that the parent pointers serve as alternative way to specify that one class inherits from another (as with *extends* in Java), but without the burden of additional syntax.

Polymorphism

Since your language is dynamically, not statically typed, polymorphism comes for free and you do not need to do anything else to implement it.