Objects

Lecture 21 Tuesday, April 16, 2013

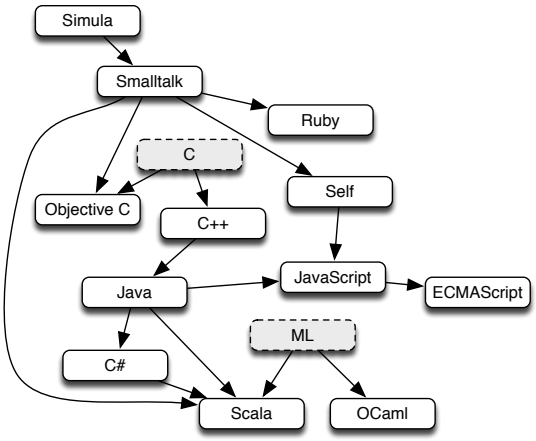
1 Objects and Object-Oriented Programming

An *object* is an entity with both data and code. Typically, objects encapsulate some or all of their data and functionality. That is, they hide away details of their implementation, and instead present some simplified view to clients. A key idea of objects is that they provide a uniform way to encapsulate arbitrary functionility/complexity: objects can represent very simple combinations of data and functionality (such as an integer, or a string), to very complex combinations (such as a database).

The data encapsulated by an object are typically called *fields* (also known as *member variables*, or *instance variables*), and the procedures of an object are typically called *methods* (aka *member functions*). An object-oriented program consists of different objects, that interact with each other by calling methods, also known as sending messages between objects.

Objects were originally introduced in the Simula programming language, in the 1960s, as a means to help write simluation programs. That is, objects were used as a convenient way to model the behavior of real-world entities. This is perhaps one of the reasons for the success of object-oriented design and programming: objects provide a way to structure and model the interaction of potentially complex systems, and varying levels of abstraction.

Many languages have object-oriented features. The following diagram shows some object oriented languages, with some indications of genealogy. (The diagram is by no means exhaustive. Genealogy information should be taken with a grain of salt; ML and C are shown in the diagram but are not OO languages.)



1.1 Object-oriented concepts

Mitchell descirbes four basic concepts of object-oriented programming languages:

1. **Dynamic lookup (aka dynamic dispatch).** When a message is sent to an object, the code to execute is determined by the way the object is implemented, and not by some static property of the pointer to the object. That is, different objects can respond to the same message in different ways. In Java and Smalltalk, method dispatch is dynamic (except for static methods in Java). In C++, only virtual member functions are selected dynamically.
2. **Abstraction.** Objects can hide implementation details from clients. This is similar to existential types, which could be used to hide both code and data from clients. In Java, fields and methods can be declared `private`, meaning that those fields and methods are accessible only within the implementation,

and not to clients of an object. In some object-oriented languages, all fields are private, and can only be accessed through public methods.

3. **Subtyping.** Like we have seen previously, if an object x has all of the (public) functionality of an object y, then object x should be able to be used wherever object y can be used. Subtyping provides both re-use and extensibility. That is, subtyping permits uniform operations over different objects, and in addition, allows us to extend the functionality of a program by adding new objects that are subtypes of existing objects.

4. **Inheritance.** Inheritance allows new objects to be defined from existing objects. While this can be seen as a form of code re-use, it also provides an extension mechanism: by inheriting code, but overriding some of the code, we can change the behavior of existing objects.

   Note that inheritance is not the same as subtyping. We can think of subtyping as a relation on the interfaces that objects present to clients, and think of inheritance as a relation on the implementation of objects. In Java, these notions are conflated, but C++ keeps these notions separate: a C++ class can declare a "private base class", that is, inheriting the behavior of another class, but not revealing that fact to clients.

1.2 Classes vs. prototypes

A class is a construct that is used to describe code that is common to all objects of that class. Different objects of the same class differ just in their state: all their code is common. Classes can be used to create new instances, i.e., new objects of that class. Languages like Java and C++ use classes.

By contrast, in a prototype-based language, there are no classes. Instead, inheritance is achieved by cloning exisiting objects, known as prototypes. Prototypes can be used to express class-like mechanisms: instead of defining a template for an object (i.e., like a class), one can create, define, and specialize an object first, and then treat it as a prototype for others like it, cloning the prototype as needed. Delegation can be used to efficiently implement cloning: instead of creating a complete copy of the prototype, the new object can simply delegate all messages to the prototype. The new object can be modified (adding or overriding fields and methods) to specialize behavior. JavaScript is a prototype-based language.

2 Object encodings

We've just informally described object-oriented concepts. How do these concepts relate to language features and mechanisms that we have already examined during this course?

2.1 Records

Records provide both dynamic lookup and subtyping. For dynamic lookup, given value v of record type, the expression $v.l$ will evaluate to a value determined by v, not by the record type. If $v.l$ is a function, then this is like dynamic dispatch: the code to invoke depends on the object v.

Moreover, we defined subtyping on record types, which permits both reuse and extension: code that expects a value of type τ can be re-used with a value of any subtype of τ; new subtypes can be created, allowing code to be extended.

Recursive records allow us to express records that can perform functional updates.

For example, consider a representation of a 2 dimensional point, which has a method to move the point in one dimension.

$$\begin{array}{l}\mathsf{letrec\ new} = \lambda i.\, \lambda j.\, \mathsf{fix\ this}.\, \{\, \mathsf{x} = i, \mathsf{y} = j, \mathsf{mvx} = \lambda d.\, \mathsf{new}\ (\mathsf{this.x} + d)\ \mathsf{this.y}\ \} \\ \mathsf{in}\ (\mathsf{new}\ 0\ 0).\mathsf{mvx}\ 10\end{array}$$

In this example, we use a recursive function new to construct a record value. The record contains fields x and y, which record the point's coordinates, and a method mvx, which takes a number d as input, and returns a point that is d units to the right of the original point. In order to construct the new point, the method

mvx calls the function new, and gives it the original x coordinate plus d, and the original y coordinate. To access the original coordinates, the method mvx must access the fields x and y of the record of which it is a field; we achieve this by using a fix point operator on the record, with the variable name this being used to refer (recursively) to the record.

2.2 Existential types

Existential types can be used to enforce abstraction and information hiding. We saw this last lecture, when we considered a simple module mechanism based on existential types, which allowed the module to export an interface that abstracted the implementation details.

2.3 Other encodings

It is possible to combine recursive records and existential types: see "Comparing Object Encodings", by Bruce, Cardelli, and Pierce, Information and Computation 155(1/2):108–133, 1999. However, rather than encoding objects on top of the lambda calculus, it is possible to directly define object calculi, simple languages that serve as a foundation for object-oriented languages (A Theory of Objects, by Abadi and Cardelli, Springer 1996).

Let's examine two object-oriented calculi. One is a model for JavaScript programs, the other a model of Java programs.

3 The Essence of JavaScript

Let's consider a calculus that captures the essence of JavaScript with the exception of eval commands. This calculus is λ_{JS}, developed by Guha et al.[1] at Brown University.

JavaScript is a language with a lot of quirks and surpising behavior.

Following the presentation by Guha et al., we will introduce the features of λ_{JS} incrementally.

3.1 Functions and objects

Let's initially consider a calculus with numbers, strings, booleans, functions, objects, and special values null and undefined.

$$
\begin{array}{l}
c ::= n \mid s \mid b \mid \mathsf{null} \mid \mathsf{undefined} \\
n \in \mathbb{Z}, \quad s \in \textbf{String} \quad b \in \textbf{Boolean} \\
v ::= c \mid \mathsf{func}(x_1, \ldots, x_n)\{\ \mathsf{return\ e}\ \} \mid \{\ s_1 : v_1, \ldots, s_n : v_n\ \} \\
e ::= x \mid v \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid e(e_1, \ldots, e_n) \mid e_1[e_2] \mid e_1[e_2] = e_3 \mid \mathsf{delete}\ e_1[e_2]
\end{array}
$$

Value $\{\ s_1 : v_1, \ldots, s_n : v_n\ \}$ is an object with fields $s_1, \ldots, s_n$, and field s_i has value v_i.

Expression $e_1[e_2]$ evaluates e_1 to an object, e_2 to a string f, and the accesses field f of the object. JavaScript allows field names to be dynamically computed. You can think of a normal field access $e.x$ as being syntactic shorthand for $e[\text{"}x\text{"}]$. Expression $e_1[e_2] = e_3$ updates a field of an object, and delete $e_1[e_2]$ deletes a field from an object.

$$
\begin{array}{l}
E ::= [\cdot] \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid E(e_1, \ldots, e_n) \mid v(v_1, \ldots, v_{i-1}, E, e_{i+1}, \ldots, e_n) \\
\quad \mid \{\ s_1 : v_1, \ldots, s_{i-1} : v_{i-1}, s_i : E, s_{i+1} : v_{i+1}, \ldots, s_n : v_n\ \} \\
\quad \mid E[e] \mid v[E] \mid E[e_2] = e_3 \mid v_1[E] = e_3 \mid v_1[v_2] = E \mid \mathsf{delete}\ E[e] \mid \mathsf{delete}\ v[E]
\end{array}
$$

[1] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In Proceedings of the 24th European Conference on Object-Oriented Programming, 2010.

$$\frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e \hookrightarrow e\{v/x\}}$$

$$\frac{}{(\mathsf{func}(x_1, \ldots, x_n)\{\ \mathsf{return\ e}\ \})(v_1, \ldots, v_n) \hookrightarrow e\{v_1/x_1\} \ldots \{v_n/x_n\}}$$

$$\frac{}{\{\ s_1 : v_1, \ldots, s_n : v_n\ \}[s_i] \hookrightarrow v_i} \qquad \frac{s \notin \{s_1, \ldots, s_n\}}{\{\ s_1 : v_1, \ldots, s_n : v_n\ \}[s] \hookrightarrow \mathsf{undefined}}$$

$$\frac{}{\{\ s_1 : v_1, \ldots, s_i : v_i, \ldots, s_n : v_n\ \}[s_i] = v \hookrightarrow \{\ s_1 : v_1, \ldots, s_i : v, \ldots, s_n : v_n\ \}}$$

$$\frac{s \notin \{s_1, \ldots, s_n\}}{\{\ s_1 : v_1, \ldots, s_n : v_n\ \}[s] = v \hookrightarrow \{\ s : v, s_1 : v_1, \ldots, s_n : v_n\ \}}$$

$$\frac{}{\mathsf{delete}\ \{\ s_1 : v_1, \ldots, s_i : v_i, \ldots, s_n : v_n\ \}[s_i] \hookrightarrow \{\ s_1 : v_1, \ldots, s_{i-1} : v_{i-1}, s_{i+1} : v_{i+1}, \ldots, s_n : v_n\ \}}$$

$$\frac{s \notin \{s_1, \ldots, s_n\}}{\mathsf{delete}\ \{\ s_1 : v_1, \ldots, s_n : v_n\ \}[s] \hookrightarrow \{\ s_1 : v_1, \ldots, s_n : v_n\ \}}$$

Some interesting things to note about JavaScript's behavior: we don't get stuck if we access a field of an object that doesn't exist. Instead, it returns the value undefined. Also, field update expression $e_1[e_2] = e_3$ can also be used to create a new field. Also, deleting a field that doesn't exist is fine: we don't get stuck.

3.2 Mutable state

In our initial calculus above, there is no mutable state: objects and variables are pure. We add first-class reference to allow mutable objects and imperative local variables to be modeled. The syntax and semantics are similar to what we have used in class previously.

$$\begin{aligned}
v &::= \cdots \mid l \\
l &\in \textbf{Locations} \\
\sigma &\in \textbf{Locations} \rightharpoonup \textbf{Values} \\
e &::= \cdots \mid e_1 := e_2 \mid \mathsf{ref}\ e \mid !e \\
E &::= \cdots \mid E := e \mid v := E \mid \mathsf{ref}\ E \mid !E
\end{aligned}$$

$$\frac{e \hookrightarrow e'}{\langle E[e],\, \sigma\rangle \longrightarrow \langle E[e'],\, \sigma\rangle} \qquad \frac{}{\langle E[\mathsf{ref}\ v],\, \sigma\rangle \longrightarrow \langle E[l],\, \sigma[l \mapsto v]\rangle}\ l \notin \mathrm{dom}(\sigma) \qquad \frac{}{\langle E[!l],\, \sigma\rangle \longrightarrow \langle E[\sigma(l)],\, \sigma\rangle}$$

$$\frac{}{\langle E[l := v],\, \sigma\rangle \longrightarrow \langle E[v],\, \sigma[l \mapsto v]\rangle}$$

Arrays in JavaScript are actually objects whose fields are called "1", "2", "3", etc. This can lead to some unexpected behavior in JavaScript.

3.3 Prototypes

In prototype-based object-oriented languages, objects can delegate their behavior to "prototypes". Typically, prototypes are just objects, but are not used to store any particular state. Instead prototype objects define structure and behavior that are common to some group of objects.

JavaScript is a prototype-based language. For example, in the following code, `car` is the prototype of `mycar`. The prototype of an object is indicated using the special field name __proto__.

```
var car = { wheels:4 }
var mycar = { __proto__:car, name:"Fenry", model:"Honda Civic", year:1986 }
```

Prototypes affect field lookup, but not field update. That is, when looking up a field in an object, if the field is not found, then the prototype will be examined, if the prototype exists. Field update, however, just directly updates the object.

We assume that the __proto__ field of an object is a reference to an object, and re-define the rules for field lookup as follows.

$$\frac{}{\{\ s_1 : v_1, \ldots, s_n : v_n\ \}[s_i] \hookrightarrow v_i} \qquad \frac{s \notin \{s_1, \ldots, s_n\} \quad \text{“__proto__”} \notin \{s_1, \ldots, s_n\}}{\{\ s_1 : v_1, \ldots, s_n : v_n\ \}[s] \hookrightarrow \mathsf{undefined}}$$

$$\frac{s \notin \{s_1, \ldots, s_n\} \quad \exists i. s_i = \text{“__proto__”} \wedge v_i = \mathsf{null}}{\{\ s_1 : v_1, \ldots, s_n : v_n\ \}[s] \hookrightarrow \mathsf{undefined}} \qquad \frac{s \notin \{s_1, \ldots, s_n\} \quad \exists i. s_i = \text{“__proto__”} \wedge v_i = l}{\{\ s_1 : v_1, \ldots, s_n : v_n\ \}[s] \hookrightarrow (!l)[s]}$$

3.4 Relationship between JavaScript and λ_{JS}

JavaScript is signficantly more complicated than λ_{JS}. There are many details that this calculus glosses over. For example, in methods of objects, there is an implicit variable called this, which refers to the current object. Also, functions are actually objects, and can have fields. Also, JavaScript has a keywork new that allows new objects to be created from functions. For example, in the following code, the line `new Point(50, 100)` executes the function `Point` with this bound to a new object, and moreover sets the __proto__ field of the new object to the prototype field of the function/object `Point`.

```
function Point(x, y) {
  this.x = x;
  this.y = y
}
pt = new Point(50, 100)
```

There are many other features of JavaScript that are not directly handled by this caluclus, including many control-flow constructs, global variables, some unintuitive behavior regarding scope of local variables, implicit coercions between types (e.g., from numbers to strings, and vice-versa).

Guha et al. provide a traslation function from JavaScript to λ_{JS}. It is not possible to prove that this translation is correct without providing a full formalization of all of JavaScript. However, they have used test suites to show that the behavior of the translation function and the λ_{JS} evaluation is consistent with three different JavaScript implementations (SpiderMonkey, V8, and Rhino).