Object-oriented concepts

Lecture 22 Thursday, April 15, 2010

See Mitchell Chapter 10 for content covered in the first part of lecture. An electronic copy of the textbook is available through the Harvard Library website. See the course web page for details.

1 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?

1.1 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.

1.2 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.

1.3 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). We now consider a simple object calculus.

2 The ς calculus

We present the sigma calculus, a pure untyped object calculus, from A Theory of Objects, by Abadi and Cardelli.

The syntax of the language is given by the following grammar.

$$e ::= x \mid [l_i = \varsigma x_i.\ e_i^{i\in 1..n}] \mid e.l \mid e_1.l \Leftarrow \varsigma x.\ e_2$$
$$v ::= [l_i = \varsigma x_i.\ e_i^{i\in 1..n}]$$

The expression x is a variable; the expression $[l_i = \varsigma x_i.\ e_i^{i\in 1..n}]$ is an object with n methods, names $l_1, .., , l_n$, with each name distinct. A method is written $\varsigma x.\ e$, where the argument x is the self parameter (i.e., it will only be replaced with the object that the method is part of), and e is the method body. The expression $v.l$ is method invocation, invoking method l on the object v. Finally, $v.l \Leftarrow \varsigma x.\ e$ updates the method l of object o with method $\varsigma x.\ e$.

Note that the order of methods in an object does not matter. Also, we can model fields as $\varsigma x.\ v$, where v doesn't mention x. For example, if we had integers, we could write $\varsigma x.\ 42$. Indeed, we will write $e_1.l := e_2$ as shorthand for $e_1.l \Leftarrow \varsigma y.\ e_2$ for some $y \notin FV(e_2)$.

We define a large step operational semantics for the ς calculus. There are just three axioms and rules.

$$\textsc{Object}\ \frac{}{v \rightsquigarrow v} \qquad \textsc{Select}\ \frac{e \rightsquigarrow v \quad v \equiv [l_i = \varsigma x_i.\ e_i^{i\in 1..n}] \quad b_j\{v/x_j\} \rightsquigarrow v'}{e.l_j \rightsquigarrow v'}\ j \in 1..n$$

$$\textsc{Update}\ \frac{e_1 \rightsquigarrow [l_i = \varsigma x_i.\ e_i^{i\in 1..n}]}{e_1.l_j \Leftarrow \varsigma x.\ e_2 \rightsquigarrow [l_j = \varsigma x.\ e_2, l_i = \varsigma x_i.\ e_i^{i\in (1..n)-\{j\}}]}\ j \in 1..n$$

Capture avoiding substitution is defined in a straightforward way, given that $\varsigma x.\ e$ binds variable x in e.

$$\begin{aligned}
\varsigma y.\ e\{v/x\} &= \varsigma y'.\ e\{y'/y\}\{v/x\} && \text{for } y' \notin FV(\varsigma y.\ e) \cup FV(v) \cup \{x\}\\
x\{v/x\} &= v\\
y\{v/x\} &= y && \text{if } y \neq x\\
[l_i = \varsigma x_i.\ e_i^{i\in 1..n}]\{v/x\} &= [l_i = (\varsigma x_i.\ e_i)\{v/x\}^{i\in (1..n)}]\\
(e.l)\{v/x\} &= e\{v/x\}.l\\
(e_1.l \Leftarrow \varsigma x.\ e_2)\{v/x\} &= e_1\{v/x\}.l \Leftarrow (\varsigma x.\ e_2)\{v/x\}
\end{aligned}$$

The sigma calculus is as expressive as the lambda calculus: we can translate from the lambda calculus as follows.

$$\begin{aligned}
[\![x]\!] &= x\\
[\![\lambda x.\ e]\!] &= [\ arg = \varsigma x.\ x.arg, val = \varsigma x.\ ([\![e]\!]\{x.arg/x\})\ ]\\
[\![e_1\ e_2]\!] &= ([\![e_1]\!].arg := [\![e_2]\!]).val
\end{aligned}$$

The following example program in the sigma calculus is intended to emulate a calculator. (We use lambda expressions in this example, since lambda expressions can be encoded.)

$$\begin{aligned}
calculator \triangleq\ & [arg = 0.0,\\
& acc = 0.0,\\
& enter = \varsigma s.\ \lambda n.\ s.arg := n,\\
& add = \varsigma s.\ (s.acc = s.equals).equals \Leftarrow \varsigma s'.\ s'.acc + s'.arg,\\
& equals = \varsigma s.\ s.arg]
\end{aligned}$$

We can use this calculator in the following ways.

$$calculator.enter(5.0).equals \rightsquigarrow 5.0$$
$$calculator.enter(5.0).add.enter(3.5).equals \rightsquigarrow 8.5$$
$$calculator.enter(5.0).add.add.equals \rightsquigarrow 15.0$$