CS411 Object Calculus

A. Demers

14 May 2001

What the lambda calculus is for traditional function-based programming languages, the object calculus is for OO languages. In these Notes we present the untyped and simply typed versions of Abadi and Cardelli's object calculus. We relate these to the lambda calculus and to the OO language features we discussed previously in an IMPX context.

1 Syntax and Evaluation Rules

The syntax of the untyped object calculus is as follows:

$$\begin{aligned} e \ ::= \ & x \\ & \mid [\ \ldots\ m_i \sim \varsigma(x_i)e_i\ \ldots\ ] \\ & \mid e.m \\ & \mid e.m \Leftarrow \varsigma(x)e \end{aligned}$$

Informally, an object consists of a tuple of names *methods*. Each method has the form

$$m \ \sim \ \varsigma(x)e$$

with the $\varsigma(x)$ portion serving as a binding occurrence of the symbol x. The $\varsigma(x)$ indicates functional abstraction, much like $\lambda\ x$ in the lambda calculus. However, a method invocation $e.m$ does not bind x to an arbitrary argument expression, as function application does in the lambda calculus. In fact, there is no argument expression; the parameter x is simply bound to the object from which the method was selected. The final form,

$$e.m \Leftarrow \varsigma(x)e$$

constructs a new object identical to the object denoted by e except that the method m has been replaced as specified.

Based on that intuitive description, here is a simple set of evaluation rules:

Values

$$\frac{}{\langle v, \phi \rangle \ \rightarrow\ v}$$

A value reduces to itself in any environment.

Identifiers

$$\frac{}{\langle x, \phi \rangle \ \rightarrow\ v} \quad \text{if } (x \sim v) \in \phi$$

An identifier is evaluated by looking it up in the environment.

Method invocation

$$\frac{\langle e, \phi \rangle \ \rightarrow\ [\ldots m_j \sim \varsigma(x_j)e_j,\ \ldots] \ \equiv\ v' \qquad \langle e_j, \phi \oplus \{x_j \sim v'\} \rangle \ \rightarrow\ v}{\langle e.m_j, \phi \rangle \ \rightarrow\ v}$$

To evaluate a method invocation, first evaluate the containing object expression e to obtain an object value v'. Then evaluate the method body e_j in an environment in which the self parameter x_j has been bound to v'.

Method update

$$\frac{\langle e, \phi \rangle \ \rightarrow\ [\ldots m_j \sim \varsigma(x_j)e_j,\ \ldots]}{\langle e.m_i \leftarrow \varsigma(x)e', \phi \rangle \ \rightarrow\ [m_i \sim \varsigma(x)e',\ \ldots,\ m_j \sim \varsigma(x_j)e_j,\ \ldots]} \quad j \in \{1..n\} - \{i\}$$

To update method m_i in object e, first evaluate e to produce some object value v, then construct and return a new object identical to v but with m_i replaced by $\varsigma(x)e'$.

The above rules are very natural. Unfortunately, they are very similar to the first set of rules given for IMPX, and like them give a dynamic scope interpretation of the calculus. Consider the follwing example:

$$[a \sim \varsigma(z)0,\ b \sim \varsigma(x)[a \sim \varsigma(z)1,\ c \sim \varsigma(y)(x.a),\ d \sim \varsigma(x)(x.c)].d].b$$

(Here "0" and "1" can be any pair of distinct terms.) We introduce abbreviations A and B for the inner and outer objects in the example:

$$\begin{array}{rcl} A & \equiv & [a \sim \varsigma(z)1,\ c \sim \varsigma(y)(x.a),\ d \sim \varsigma(x)(x.c)] \\ B & \equiv & [a \sim \varsigma(z)0,\ b \sim \varsigma(x)A.d] \end{array}$$

The entire term is simply

$$B.b$$

Should this term reduce to 0 or to 1? Intuitively, under a static scope interpretation it should reduce to 0, because the free occurrence of x in the body of c should be bound to the entire object B. However, in the invocation of c from within d, the evaluation rules given above allow the free occurrence of x to be captured by the parameter of d. The essential parts of the derivation are.

$$\begin{array}{l} \langle 1, \{x \sim B\} \oplus \{x \sim A\} \oplus \{y \sim A\} \oplus \{z \sim A\} \rangle \ \to\ 1 \\ \hline \langle x.a, \{x \sim B\} \oplus \{x \sim A\} \oplus \{y \sim A\} \rangle \ \to\ 1 \\ \hline \langle x.c, \{x \sim B\} \oplus \{x \sim A\} \rangle \ \to\ 1 \\ \hline \langle A.d, \{x \sim B\} \rangle \ \to\ 1 \\ \hline \langle B.b, \emptyset \rangle \ \to\ 1 \end{array}$$

The subgoal of evaluating "x.c" with $\{x \sim A\}$ at the top of the type assignment stack is where the problem arises.

We would much prefer a static scope interpretation for the object calculus. One way to achieve this is to eliminate the environment altogether, and give rules that rely on outermost substitution of *closed* terms to avoid capture and achieve static scope. A viable set of rules follows. For these rules a *value* v is any irreducible closed expression – that is, a closed object constructor – and all judgements produce only closed terms.

Values

$$\frac{}{v \ \to\ v}$$

A value reduces to itself.

Method invocation

$$\frac{\begin{array}{l} e \ \rightarrow\ v' \\ ([v'/x_j](e_j)) \ \rightarrow\ v \end{array}}{e.m_j \ \rightarrow\ v}$$
$$\text{where } v' \equiv [\ldots,\ m_i \sim \varsigma(x_i)e_i,\ \ldots]$$

To invoke method m_j from object e, first reduce e to a closed value v', then evaluate the method body e_j after substituting v' for the self parameter x_j.

Method update

$$\frac{e \ \rightarrow\ [m_1 \sim \varsigma(x_1)e_1,\ \ldots,\ m_n \sim \varsigma(x_n)e_n]}{(e.m_i \leftarrow \varsigma(x)e') \ \rightarrow\ [m_i \sim \varsigma(x)e',\ \ldots, m_j \sim \varsigma(x_j)e_j,\ \ldots]}$$
$$\text{where } j \in \{1..n\} - \{i\}$$

To update method m_i in object e, first reduce e to a closed value, then construct the new object in which m_i is replaced by $\varsigma(x)e'$. Note the requirement that the expressions in the judgement are closed implies that e' has no free variables other than x.

Intuitively these rules behave like outermost β-reduction. If we apply the rules to the example above, using the same abbreviations A anb B as before, the essential branch of the derivation tree is

$$\begin{array}{l} 0 \ \rightarrow\ 0 \\ \hline [([B/x](A))/y](([B/x](x.a))) \ \equiv\ B.a \ \rightarrow\ 0 \\ \hline [([B/x](A))/x](x.c) \ \equiv\ ([B/x](A)).c \ \rightarrow\ 0 \\ \hline [B/x](A.d) \ \equiv\ ([B/x](A)).d \ \rightarrow\ 0 \\ \hline B.b \ \rightarrow\ 0 \end{array}$$

Because all substitutions involve closed terms, capture does not occur, and the rules define a static scope semantics for the calculus.

Unfortunately, this approach does not lend itself to direct implementation in the style of the "Simple Runtime and Compilation" notes distributed earlier.

2 Relation to Untyped Lambda Calculus

There is a straightforward translation of the untyped lambda calculus into the untyped object calculus that preserves reductions (though not the η conversion rule), demonstrating that the object calculus, like the lambda calculus, is Turing-complete.

Recall the syntax of the lambda calculus:

$$e \quad ::= \quad x \;\mid\; \lambda x.e \;\mid\; e_1(e_2)$$

Intuitively, a lambda term $\lambda x.e$ and a method $\varsigma(x)e$ are quite similar functional abstractions. The essential difference is the restriction on the parameter imposed by the object calculus. In an invocation of a lambda term, the parameter x may be bound to an arbitrary term. In an invocation of a method, the parameter x may be bound only to the object from which the method was selected. Fortunately, it is easy to get around this apparent limitation, by exploiting the "tupling" capability of the object calculus. In the translation of a lambda application into object calculus, we simply use object update to store the parameter value in a well known attribute of the translated object. The method body can access the argument by selection from the method's self parameter.

We define the translation $T[\![\cdot]\!]$ of untyped lambda terms into terms of the untyped object calculus by induction on the structure of lambda terms as follows:

Identifiers

$$T[\![x]\!] \quad = \quad x$$

Identifiers are unchanged.

Lambda

$$T[\![\lambda x.e]\!] \quad = \quad [\; a \sim \varsigma(x)x.a,\;\; v \sim \varsigma(x)([x.a/x](T[\![e]\!]))\;]$$

A lambda term is mapped to an object with attributes a (which will eventually hold the *a*rgument value) and v (a method that will be invoked to compute the *v*alue of an application). These are used cooperatively in the translation of an application.

Application

$$T[\![e_1(e_2)]\!] \quad = \quad ((T[\![e_1]\!]).a \Leftarrow \varsigma(y)T[\![e_2]\!]).v$$

> As discussed above, the translation of an application constructs the translation of the function part e_1, then updates the a method to a ς term that will evaluate to the translation of the argument e_2. Finally, it invokes the v method to produce the translation of the result value.

Note the a attribute is not absolutely necessary in the translation of a lambda term. The body of v does include a reference to $x.a$, but no translated lambda term ever invokes the v method directly from a translated lambda term. That is, no invocations have the form

$$(T[\![e]\!]).v$$

Instead, the invocations are all of the form

$$((T[\![e_1]\!]).a \Leftarrow \varsigma(y)T[\![e_2]\!]).v$$

It is perfectly all right to "update" the a attribute of an object that does not (yet) have an a attribute. So by the time the v method is invoked an a attribute will exist. Nevertheless, we leave the a attribute in the translation of a lambda term, so later we can use the translation rules for the typed system.

Correctness of the above translation is shown by the following theorem, which can be proved by induction on derivations.

Theorem. Let e_1 and e_2 be terms in the untyped lambda calculus, and let T be the translation of untyped lambda calculus terms into untyped object calculus as defined above. Then

$$T[\![e_1]\!] \twoheadrightarrow_o T[\![e_2]\!] \qquad \Leftrightarrow \qquad e_1 \twoheadrightarrow_\lambda e_2$$

where the reduction relations $\twoheadrightarrow_o$ and $\twoheadrightarrow_\lambda$ refer to the object and lambda calculi, respectively. □

3 Functional vs Imperative Interpretations

In most real-world OO languages, attribute update is imperative. In a method update expression of the form

$$(e_1.m \leftarrow \varsigma(x)e_2)$$

the object denoted by e_1 is updated "in place" – subsequent invocations of the m method of the original object will see the new definition. In contrast, we have defined the object calculus *functionally* – that is, without side effects. The method update expression above produces an entirely new object with the new definition of m; subsequent invocations of the m method of the original object continue to use the original definition.

3.1 Distinguishable

With lazy reduction rules it is not altogether trivial to write an expression that demonstrates the difference between these two strategies; but it *is* possible. Consider the term

$$[\ a \sim \varsigma(x)A,\ \ b \sim \varsigma(x)((x.a \leftarrow \varsigma(y).x.a).a)].b$$

where A is an arbitrary convergent term. Using the static scope reduction rules given above, which are functional, this expression reduces to A. With an imperative interpretation, however, the expression would diverge. We can argue this only informally, since we haven't given a formal semantics for the imperative case. The argument goes like this: in the invocation of the b method, both instances of x in

$$((x.a \leftarrow \varsigma(y).x.a).a)$$

are bound to the same (initial) object. After executing the (imperative) update of method a, the state looks like

$$x \ \sim \ [\ a \sim \varsigma(y)x.a\ \ b \sim \varsigma(x)((x.a \leftarrow \varsigma(y).x.a).a)]$$

The body of $x.a$ is an invocation of $x.a$, and thus the evaluation diverges.

3.2 Imperative is Order Dependent

Intuitively, attribute update "in place" seems similar in power to an assignable store. Recall that assignments in IMPX make the result of evaluation more dependent on evaluation order – specifically, they make it possible for a convergent expression to produce different values under eager and lazy evaluation rules. A similar phenomenon obtains in the imperative object calculus. For this example, we extend the calculus with **let** expressions, as they make the example a lot shorter. Let A and B be any two distinct object calculus terms. Consider

$$\begin{array}{l} \textbf{let } x \ \sim \ [a \sim \varsigma(s)A] \textbf{ in} \\ \quad \textbf{let } y \ \sim \ x.a \leftarrow \varsigma(s)B \textbf{ in} \\ \quad\quad x.a \end{array}$$

With an imperative semantics, evaluation of the expression bound to y will update the object bound to x "in place." Thus, if *let* blocks are evaluated eagerly, the final result $x.a$ will see the effect of the inner *let* block expression, and produce the value of B. Using lazy evaluation, the bound expression of the inner *let* block will never be evaluated at all (since y is not referenced), and the result value will be A.