CS611 Lecture 38: Objects: classes, advanced features — November 29, 1999
Scribe: Nate Nystrom — Lecturer: Andrew Myers

1 Untyped Object Calculus

Recall our expression for constructing a record:

$$\{\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_n = e_n\}$$

If we are to use this record as an object, we wish to be able to refer to the object within the expressions $e_1, e_2, \ldots, e_n$. Thus we might try

$$\textsf{rec this } \{\ell_1 = e_1, \ell_2 = e_2, \ldots, \ell_n = e_n\}$$

But we saw last time that if we modify the object by adding or changing a field, this still refers to the old version of the object. Thus, we define a language to include a new identifier for each field of an object with which we can refer to the object. The syntax of *untyped object calculus* is:

$$e ::= \{x_1.\ell_1 = e_1, x_2.\ell_2 = e_2, \ldots, x_n.\ell_n = e_n\} \mid x \mid e.\ell \mid e_0 \textsf{ with } x.\ell = e_1$$

The identifiers x_i in the object expression $\{x_1.\ell_1 = e_1, x_2.\ell_2 = e_2, \ldots, x_n.\ell_n = e_n\}$ are bound so that they refer to the object when used within e_i. For example, if we define the a point object as

$$\textsf{point} \equiv \{\textsf{p.x} = 1, \textsf{p}'\textsf{.y} = 1, \textsf{p}''\textsf{.get\_x} = \textsf{p}''\textsf{.x}, \textsf{p.move\_x} = \lambda \textsf{d}.\ \textsf{p with p}'\textsf{.x} = \textsf{p.x} + \textsf{d}\}$$

then $\textsf{p}$, $\textsf{p}'$, and $\textsf{p}''$ all refer to the object point.

1.1 Representing Functions

Note that the untyped object calculus is as powerful as the untyped lambda calculus. We can encode a function in the lambda calculus as an object with two fields, val and arg, as follows:

$$\begin{aligned} [\![\lambda x.\, e]\!] &= \{x.\textsf{val} = [\![e]\!][x.\textsf{arg}/x]\} \\ [\![(e_0\ e_1)]\!] &= ([\![e_0]\!] \textsf{ with } x.\textsf{arg} = [\![e_1]\!]).\textsf{val} \end{aligned}$$

Using this desugaring, we can even define a fixed point operator:

$$Y = \{x.\textsf{arg} = x.\textsf{arg}, x.\textsf{val} = (x.\textsf{arg with } x.\textsf{arg} = x.\textsf{val}).\textsf{val}\}$$

1.2 Operational Semantics

Define

$$o \equiv \{x_1.\ell_1 = e_1, x_2.\ell_2 = e_2, \ldots, x_n.\ell_n = e_n\}$$

Then we have the following operational semantics for the expressions in the object calculus

$$\frac{}{o.\ell_i \longrightarrow_1 e_i[o/x_i]}$$

$$\frac{}{o \textsf{ with } y.\ell = e \longrightarrow_1 \{y.\ell = e, x_i.\ell_i = e_i \forall i.\ell_i \neq \ell\}}$$

The other rules as as one would expect, e.g.,

$$\frac{e \longrightarrow_1 e'}{e.\ell \longrightarrow_1 e'.\ell}$$

2 Classes

The untyped object calculus is an object-based language. To model a class-based languages, we can consider a class to be a set of functions that produce objects.

$$[\![\text{class } c\ x_i.\ell_i = e_i \text{ end}]\!] = \lambda u.\, \{x_i.\ell_i = e_i\}$$

For brevity we use $x_i.\ell_i = e_i$ as shorthand for

$$x_1.\ell_1 = e_1, x_2.\ell_2 = e_2, \ldots, x_n.\ell_n = e_n$$

We can think of $\lambda u.\, \{x_i.\ell_i = e_i\}$ as a constructor for objects of type c. We invoke the constructor with the new operator:

$$[\![\text{new } c]\!] = (c\ \#\text{u})$$

To implement (single) inheritence, we define

$$[\![\text{class } c \text{ extends } c'\ x_i.\ell_i = e_i \text{ end}]\!] = \lambda u.\, (c'\ \#\text{u}) \text{ with } x_i.\ell_i = e_i$$

This constructor calls the superclass constructor and then extends the object with the fields in the subclass. For multiple inheritence, we define

$$[\![\text{class } c \text{ extends } c', c'', \ldots, c^{(n)}\ x_i.\ell_i = e_i \text{ end}]\!] = \lambda o.\, (c'\ (c''\ \ldots\ (c^{(n)}\ o)\ \ldots)) \text{ with } x_i.\ell_i = e_i$$

with the root class case

$$[\![\text{class } c\ x_i.\ell_i = e_i \text{ end}]\!] = \lambda o.\, o \text{ with } x_i.\ell_i = e_i$$

and the new operator

$$[\![\text{new } c]\!] = (c\ \{\})$$

In this implementation the order in which the base classes are listed is important.

2.1 Virtual and non-virtual inheritence

In languages that support multiple inheritence the question arises of how to handle a diamond in the class hierarchy:

$$\begin{aligned}
&\text{class } A\ \ldots \text{ end}\\
&\text{class } B \text{ extends } A\ \ldots \text{ end}\\
&\text{class } C \text{ extends } A\ \ldots \text{ end}\\
&\text{class } D \text{ extends } B, C\ \ldots \text{ end}
\end{aligned}$$

C++ has two ways of implementing inheritence.

1. In "non-virtual inheritence", a derived class has a separate copy of the fields of a base class for each time it appears as a base class. In the example above, since D inherits A twice, once through B and once through C, two separate copies of the fields of A are stored in an object of class D.

2. With "virtual inheritence", only one copy of the fields of a base class is stored in an object of the derived class. In the example, only one copy of the fields of A is stored in an object of class D.

2.2 Abstract classes and traits

A class can be defined that creates incomplete objects. Such a class is called an *abstract class*. In languages such as Java and C++ abstract classes cannot be instantiated. In object-based languages, an incomplete object is called a *trait*. Abstract classes are useful for defining interfaces.

2.3 Mixins

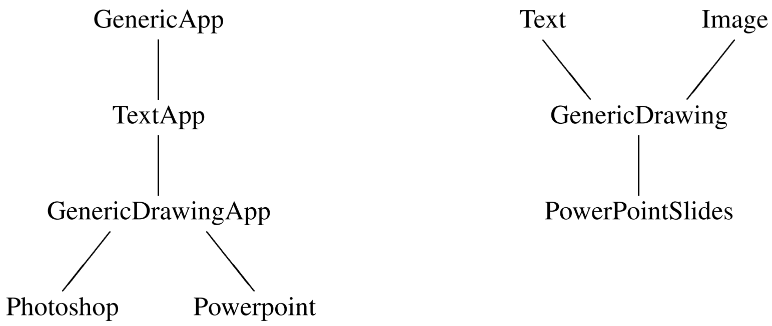
A class whose superclass is a parameter is called a *mixin*. A mixin is often used to provide an optional interface or functionality to another class. A mixin can be used to simulate multiple inheritence. We can implement mixins in the object calculus as

$$[\![\mathsf{mixin}\ c\ x_i.\ell_i = e_i\ \mathsf{end}]\!] = \lambda c'.\,\lambda o.\,(c'\ o)\ \mathsf{with}\ x_i.\ell_i = e_i$$

2.4 Multimethods

A *multimethod* or a *generic function* is a method with multiple receivers per invocation. For example, $(o_1, o_2).\ell$. Languages which support multimethods include CLOS and Cecil.

As an example, imagine a set of application classes and data classes with the hierarchies below.



We might implement the following multimethods:

- $(\mathsf{TextApp}, \mathsf{Text}).\mathsf{receive}$
- $(\mathsf{Photoshop}, \mathsf{GenericDrawing}).\mathsf{receive}$
- $(\mathsf{Powerpoint}, \mathsf{GenericDrawing}).\mathsf{receive}$
- $(\mathsf{Powerpoint}, \mathsf{PowerPointSlides}).\mathsf{receive}$

When a multimethod is invoked, the most specific implementation is called. So if we invoke

$$(\mathsf{Photoshop}, \mathsf{PowerPointSlides}).\mathsf{receive},$$

the implementation

$$(\mathsf{Photoshop}, \mathsf{GenericDrawing}).\mathsf{receive}$$

is called.

Note that there may be more than one most specific implementation. If

$$(\mathsf{GenericDrawingApp}, \mathsf{PowerPointSlides}).\mathsf{receive}$$

were implemented than the invocation of

$$(\mathsf{Photoshop}, \mathsf{PowerPointSlides}).\mathsf{receive}$$

could call either

$$(\mathsf{Photoshop}, \mathsf{GenericDrawing}).\mathsf{receive}$$

or

$$(\mathsf{GenericDrawingApp}, \mathsf{PowerPointSlides}).\mathsf{receive}.$$

Different languages handle such ambiguities differently.