Object-Oriented Programming

Lecture 15
CS 565
4/3/06
Object-Based Programming

- View basic features found in object-based systems:
  - objects
  - dynamic dispatch
  - encapsulation of state
  - inheritance
  - self-reference (self/this)
  - late binding

- As derived forms in a lower-level language:
  - records
  - references
  - recursion
  - subtyping
Characterization

- Dynamic dispatch:
  - when an operation is invoked on an object, the resulting actions depend upon the object itself.
  - different objects which respond to the same set of operations (interface/signatures) may have different implementations
  - determining the actions to be performed may not be possible at compile-time
    - contrast with functions and their application
Characterization

- Encapsulation
  - Objects consist of both internal and external state
  - Not all “oo”-languages provide this kind of encapsulation (e.g., Dylan, Cecil, or CLOS)
  - Originally used as a form of information hiding (CLU or Simula 67)
    - hidden representation type t
    - a collection of operations for manipulating t
    - only one hidden representation and only one implementation
    - commonly referred to as abstract data types
Subtyping

- The type of an object is the set of operations that can be performed on it.
- An interface listing more operations is “better” than one listing fewer operations.
- Types do not expose internal representation.
- Object interfaces fit naturally into a subtype model.
Inheritance

- Objects that share parts of their interfaces can sometimes share parts of their behavior.
  - Avoid code duplication
  - Code reuse expressed via a class structure and form of subclassing that allows new classes to be derived from old ones by adding implementation features
Late Binding

- Allows a method within a class to call another method via the pseudo-variable (self or this). If the second method is overridden by some subclass so is the first.
Objects

Example: A simple counter

c = let x = ref 1
    in { get = λ _:Unit. !x,
        inc = λ _:Unit. x := succ(!x)
    } → c : Counter

where Counter = {get : Unit → Nat,
        inc : Unit → Unit}
Objects and Object Generators

inc3 = \( c : \text{Counter} \). ( c.inc \text{unit}, c.inc \text{unit}, c.inc \text{unit} ) \rightarrow 
\quad \text{inc3 : Counter} \rightarrow \text{unit}

(inc3 c; c.get \text{unit}) \rightarrow 4

newCounter = 
\quad \lambda _ \cdot \text{Unit}. \text{let } x = \text{ref } 1 
\quad \quad \text{in} \{ \text{get} = \lambda _\cdot \text{Unit}. !x,
\quad \quad \quad \text{inc} = \lambda _\cdot \text{Unit}. x := \text{succ}(!x)\}\}
\quad \rightarrow \text{newCounter : Unit} \rightarrow \text{Counter}
ResetCounter = { get : Unit → Nat,  
                 inc : Unit → Unit,  
                 reset : Unit → Unit}

newResetCounter =  
λ _: Unit. let x = ref 1  
in { get = λ _:Unit. !x,  
    inc = λ _:Unit. x := succ(!x),  
    reset = λ _ :Unit. x := 1}  
→ newResetCounter : Unit → ResetCounter

rc = newResetCounter unit;  
(inc3 rc; rc.reset unit; inc3 rc; rc.get unit) → 4
Classes

- Both newCounter and newResetCounter provide identical implementations except for the reset method.
- Violates basic software engineering principles:
  - each piece of behavior should be implemented in just one place in the program.
Reuse

resetCounterFromCounter = 
λ c:Counter. let x = ref 1
    in { get = c.get,
        inc = c.inc,
        reset = λ_:Unit. x := 1}

What’s wrong with this approach?
Classes

Need to separate definition of methods from state manipulated by these methods:

counterClass = \( \lambda \ r: \text{CounterRep} \) .

\[
\begin{align*}
\{ \text{get} &= \lambda \_: \text{Unit}. \! (r.x), \\
\text{inc} &= \lambda \_: \text{Unit}. r.x := \text{succ}(\! (r.x)) \}
\end{align*}
\]

\( \rightarrow \) counterClass : \text{CounterRep} \rightarrow \text{Counter}

CounterRep: \{ x: \text{Nat} \ \text{ref} \}
Subclass

resetCounterClass =
λ r: CounterRep.
  let super = counterClass r
  in { get = super.get,
       inc = super.inc,
       reset = λ :Unit. r.x := 1}

→ resetCounterClass : CounterRep → ResetCounter

newResetCounter =
λ _. Unit. let r = {x=ref 1} in resetCounterClass r

→ newResetCounter : Unit → ResetCounter
Extending Representations

- May wish to add new instance variables (fields) to a representation

BackupCounter = \{get: Unit \rightarrow \text{Nat}, \quad \text{inc: Unit \rightarrow Unit,} \quad \text{reset: Unit \rightarrow Unit,} \quad \text{backup: Unit \rightarrow Unit}\} \\
BackupCounterRep = \{x: \text{Nat ref, b: Nat ref}\}
Instance Variables

backupCounterClass =
  \ r: BackupCounterRep. 
    let super = resetCounterClass r 
    in {get = super.get, 
        inc = super.inc, 
        reset = \_: Unit. r.x := !(r.b), 
        backup = \_: Unit. r.b := !(r.x)}

→ backupCounterClass : 
  BackupCounterRep → BackupCounter

Two interesting features:
  1. overrides method reset defined in resetCounterClass
  2. uses subtyping in definition of super:
     resetCounterClass expects an argument of type CounterRep, but we
     are providing an argument of type BackupCounterRep which has
     more fields.
Invoking Super

Suppose every call to inc must first backup the current state. Avoid copying the code for backup by making inc use the backup and inc methods from super:

funnyBackupCounterClass =
  λ r: BackupCounterRep.
    let super = BackupCounterClass r
    in {get = super.get,
        inc = λ _: Unit. (super.backup unit; super.inc unit),
        reset = super.reset,
        backup = super.backup}

funnyBackupCounterClass:
  BackupCounterRep → BackupCounter