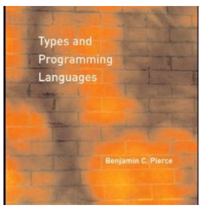
Object-Oriented Programming

Lecture 16
CS 565



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)
  - 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 the call will go to the overriding method.

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→int, inc:Unit →Unit}
```

Objects and Object Generators

```
inc2 = λc:Counter.(c.inc unit,c.inc unit)
⊢inc2: Counter → unit

(inc2 c; c.get unit) ⇒ 3


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

Subtyping

```
ResetCounter = { get : Unit → Nat,
                 inc : Unit → Unit,
                 reset : Unit → Unit}
resetCounter =
  λ _: Unit. let x = ref 1
       in { get = λ _ :Unit. !x,
            inc = λ _:Unit. x := succ(!x),
            reset = λ _ :Unit. x := 1}
resetCounter: Unit → ResetCounter
rc = resetCounter unit;
(inc2 rc;rc.reset unit;inc3 rc;rc.get unit) ⇒ 4
```

Classes

Both `newCounter` and `resetCounter` 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 = λ r:CounterRep.
          {get = λ _:Unit.!(r.x),
           inc = λ _:Unit.r.x := succ(!(r.x))}

⊢ counterClass: CounterRep → Counter
⊢ CounterRep: { x: Nat ref }
```

Subclass

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

⊢ rCounterClass: CounterRep → ResetCounter

resetCounter = λ _: Unit.
       let r = {x=ref 1} in rCounterClass r

⊢newResetCounter : Unit → ResetCounter
```

Extending Representations

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

```
BackupCounter = {get: Unit → Nat,
                 inc: Unit → Unit,
                 reset: Unit → Unit,
                 backup: Unit → Unit}
BackupCounterRep = {x:Nat ref, b:Nat ref}
```

Instance Variables

```
bCounterClass =
   λ r: BackupCounterRep.
        let super = rCounterClass r
        in {get = super.get, inc = super.inc,
            reset = λ _: Unit. r.x := !(r.b),
            backup = λ _:Unit. r.b := !(r.x)}
bCounterClass: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

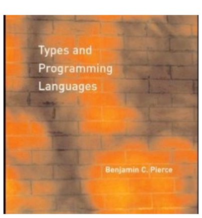
Suppose every call to `inc` must first `backup` the current state. Avoid copying the code for `backup` by making `inc` use the `backup` and `inc` methods from `super`:

```
funnyBCClass =
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}

⊢funnyBCClass: BackupCounterRep → BackupCounter
```

Self-Reference

Lecture 17
CS 565



Method invocations

Consider a class defining counters with `get`, `set`, and `inc` methods:

```
setCounterClass =
 λ r: CounterRep.
  { get = λ  _: Unit. !(r x),
    set = λ _: Unit. r.x := 1,
    inc = λ _: Unit. r.x := (succ r.x) }
```

Bad style: can express `inc` in terms of `get` and `set`

Would like to avoid repeating implementation of this functionality.

Method invocations

```
setCounterClass = λ r: CounterRep.
 fix(λ self: SetCounter.
  {get = λ_:Unit. !(r x),
   set = λ_:Unit. r.x := 1,
   inc = λ_:Unit.self.set(succ(self.get unit)})
```

The type of the inner λ abstraction is `SetCounter` → `SetCounter` so the type of the object returned by the fix expression is `SetCounter`

`setCounterClass: CounterRep` → `SetCounter`

- `SetCounter` is a record type that corresponds to the record returned by the inner abstraction.
- Define a set of mutually recursive functions.

Understanding Self

Note that the fixed point in `setCounterClass` is closed – the recursion is closed when we build the record.

```
λ r: CounterRep.
 fix(λ self: SetCounter.
  {get =λ_:Unit. !(r x),
   set =λ_:Unit. r.x := 1,
   inc =λ_:Unit. self.set(succ(self.get unit)})
```

This does not model the behavior of self or this found in real object oriented languages. Why?

Another Approach

Idea: Move the application of fix from the class definition.

```
setCounterClass =
 λr:CounterRep.
  (λ self: SetCounter.
   {get =λ_:Unit.!(r x),
    set =λi:Nat.r.x := i,
    inc =λ_:Unit.self.set(succ(self.get unit)})
```

- to the object creation function:

```
newSetCounter = λ _:Unit.
      let r = {x=ref 1}
      in fix(setCounterClass r)
```

In essence: switch the order of `fix` and `λr:CounterRep`

Types

The types have changed from:

```
setCounterClass =
 λr:CounterRep.
  fix(λ self: SetCounter.
   {get =λ_:Unit. !(r x),
    set =λ_:Unit. r.x := 1,
    inc =λ_:Unit.self.set(succ(self.get unit)})

⊢setCounterClass: CounterRep  →  SetCounter
```

Types

```
setCounterClass =
 λ r: CounterRep.
  (λ self: SetCounter.
   {get = λ: Unit. !(r x),
    set = λi: Nat. r.x := i,
    inc = λ:Unit.self.set(succ(self.get unit)})

⊢setCounterClass:
    CounterRep →SetCounter →SetCounter
```

Using Self

Consider a new class of counter objects defined to be a subclass of set-counters that keeps a record of the number of times a counter is set:

```
InstrCounter = { get : Unit → Nat,
                 set: Nat  → Unit,
                 inc: Unit  → Unit,
                 accesses: Unit → Nat}

InstrCounterRep = { x:Nat ref, a:Nat ref}
```

Implementation

```
instrCounterClass =
 λr: InstrCounterRep.
  λself: InstrCounter.
   let super = setCounterClass r self in
    {get = super.get,
     set =λi:Nat.(r.a:=succ(!r a);super.set i),
     inc = super.inc,
     accesses = λ_:Unit.!(r a)}

⊢instrCounterClass:
  InstrCounterRep →InstrCounter →InstrCounter
```

Observations

The methods in `instrCounterClass` use both `self` (passed as a parameter) and `super` (constructed using `self` and the representation)

The definition of `inc` in `super` will invoke the set and get methods defined here which in turn calls set.

Subtyping plays a crucial role here in the call to `setCounterClass` (how?)

Issues

Consider how an instance of an instrumented counter is created:

```
λ_:Unit.let r = { x = ref 1, a = ref 0}
        in fix (instrCounterClass r)
```

Problem: the construction of super happens in an "unprotected" piece of code (not encapsulated by an abstraction):

```
instrCounterClass =
  λ r: InstrCounterRep.
    λ self: InstrCounter.
     let super = setCounterClass r self in …
```

What happens here?

Example

```
ff = λ f: Nat  → Nat
            let f′ = f
            in λ n: Nat. 0
ff : (Nat  → Nat)  → (Nat  → Nat)
```

But, `fix ff  ⇒  ff (fix ff)`

```
            ⇒ let f′ = (fix ff) in ...
            ⇒ let f′ = ff (fix ff) in ...
            ⇒ ?
```

Eager Evaluation of Super

When we apply `fix (instrCounterClass r)`

- We evaluate

```
fix (λself:InstrCounterClass
       let super = setCounterClass r self in
        ...
```

- By evaluation rule for fix, this yields

```
let super = setCounterClass r (fix λself...)
in)
```

- However, to reduce the application of `setCounterClass` requires us to reduce `(fix λself...)` to a value. The current structure of the `InstrCounterClass` will not permit that. Intuitively, self is being applied to fix too early.

Remedy

Delay evaluation of self via a dummy abstraction:

```
setCounterClass =
 λ r:CounterRep.
  (λself: Unit → SetCounter.
     λ_:Unit.
      { get = λ  : Unit. !(r x),
        set = λ i: Nat. r.x := i,
        inc = λ _ : Unit.(self unit).set
                   (succ ((self unit).get unit))})

⊢ setCounterClass:
   CounterRep →(Unit →SetCounter) → SetCounter
```

Remedy

```
instrCounterClass =
 λr:InstrCounterRep.
  λself:Unit → InstrCounter.
   λ_:Unit.
    let super = setCounterClass r self unit
    in
    {get =super.get,
     set =λi:Nat.(r.a:=succ(!r a);super.set i),
     inc =super.inc,
     accesses =λ:Unit.!(r a)
newInstrCounter =
   λ_:Unit.let r = {x = ref 1, a = ref 0}
           in fix (instrCounterClass r) unit
```

Evaluation

This approach is correct in that we can instantiate `instrCounterClass` (without diverging).

However, delaying the evaluation of self has the unfortunate effect of “recomputing” the object definition everytime self is evaluated.

- Are there better approaches?

Implementing Self

The main problem with the previous approach is that methods to self are recomputed every time a call is made.

Two alternatives:

- use different implementation strategy, e.g. use references instead of fixpoints.
- Abandon the notion of encoding objects directly in the lambda calculus, developing instead an alternative calculus in which objects and classes are primitives

Using References

Intuition: instead of abstracting a record of methods that is created using fix, abstract a reference to a record of methods and allocate this record first.

```
setCounterClass =
 λ r: CounterRep.
   λ self: SetCounter ref.
     {get = λ_: Unit. !(r.x),
      set = λi: Nat. r.x := i,
      inc = λ_:Unit.
         (!self).set(succ((!self).get unit))}
```

The self parameter is a reference to a cell that contains the method of the current object.

Instantiation

To create a counter, we first create a dummy counter and then subsequently set it:

```
dummySetCounter =
   { get = λ _ : Unit. 0,
     set = λ i:Nat. unit,
     inc = λ _ : Unit. unit}'
newSetCounter = λ_:Unit.
  let r={x=ref 1}, c=ref DummySetCounter
  in (c := (setCounterClass r c); !c)
```

Since all dereferences to self are protected inside an abstraction, the contents of the dummy counter will never be accessed. What is the general problem with using references to model inheritance and subclassing?