

Lesson 8

Imperative Objects

2/12
Chapter 18

Object-Oriented Programming

- Multiple representations (“polymorphism”)
- Encapsulation of state
- Subtyping
- Inheritance (sharing implementation code)
- Open Recursion

Simple Objects

```
c = let x = ref 1 in
  {get = λ _ : Unit . !x,
   inc = λ _ : Unit . x := succ(!x)}
> c : {get : Unit -> Nat, inc : Unit -> Unit}
```

```
c.inc unit;
> unit : Unit
c.get unit;
> 2 : Nat
(c.inc unit; c.inc unit; c.get unit)
➤ 4 : Nat
```

```
type Counter = {get : Unit -> Nat, inc : Unit -> Unit}
```

Simple Objects

```
type Counter = {get : Unit -> Nat, inc : Unit -> Unit}
```

```
inc3 = λ c : Counter . (c.inc unit; c.inc unit; c.inc unit)
```

```
> inc3 : Counter -> Unit
```

```
(inc3 c; c.get unit)
```

```
> 7 : Nat
```

Object generators

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

> newCounter: Unit -> Counter

Subtyping

```
type 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 . c := succ(!x),  
     reset = λ _ : Unit . x := 1}
```

```
> newResetCounter: Unit -> ResetCounter
```

```
rc = newResetCounter unit
```

```
(inc3 rc; rc.reset unit; inc3 rc; rc.get unit);
```

```
> 4 : Nat
```

Grouping Instance Variables

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

```
> c : Counter
```

```
type CounterRep = {x : Ref Nat}
```

Simple Classes

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

> counterClass : CounterRep -> Counter

```
newCounter =  
  λ _ : Unit . let r = {x = ref 1} in counterClass r
```

> newCounter: Unit -> Counter

Simple Classes - Inheritance

```
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
```

Adding Instance Variables

```
type BackupCounter = {get : Unit -> Nat, inc : Unit -> Unit,  
                      reset : Unit -> Unit, backup : Unit -> Unit}
```

```
type BackupCounterRep = {x : Ref Nat, b : Ref Nat}
```

```
backupCounterClass =
```

```
  λ r : BackupCounterRep .
```

```
    let super = resetCounterClass r in      (* r : CounterRep *)
```

```
      {get = super.get,
```

```
        inc = super.inc,
```

```
        reset = λ _ : Unit . r.x := !(r.b),    (* override *)
```

```
        backup = λ _ : Unit . r.b := !(r.x)}
```

```
> backupCounterClass : BackupCounterRep -> BackupCounter
```

Calling Superclass methods

```
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
```

Classes with Self

```
type SetCounter = {get : Unit -> Nat, set : Nat -> Unit,  
                  inc : Unit -> Unit}
```

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

Open Recursion

```
setCounterClass =  
  λ r : CounterRep .  
    λ self : SetCounter  
      {get = λ _ : Unit . !(r.x),  
       set = λ i : Nat . r.x := i,  
       inc = λ _ : Unit . self.set(succ(self.get unit))}  
> setCounterClass : CounterRep -> SetCounter -> SetCounter  
  
newSetCounter =  
  λ _ : Unit . let r = {x = ref 1} in fix (setCounterClass r)  
> newSetCounter = Unit -> SetCounter
```

Open Recursion, Inheritance

```
type InstrCounter = {get : Unit -> Nat, set : Nat -> Unit,  
                    inc : Unit -> Unit, accesses: Unit -> Nat}
```

```
type InstrCounterRep = {x: Ref Nat, a : Ref Nat}
```

```
instrCounterClass =
```

```
  λ r : InstrCounterRep .
```

```
  λ self : InstrCounter .
```

```
    let super = setCounterClass r self in
```

```
      {get = super.get,
```

```
        set = λ i : Nat .(r.a := succ(!(r.a)); super.set i),
```

```
        inc = super.inc,
```

```
        accesses = λ _ : Unit. !(r.a)}
```

```
> instrCounterClass : InstrCounterRep -> InstrCounter -> InstrCounter
```

```
newInstrCounter = λ _ : Unit . let r = {x = ref 1, a = ref 0} in
```

```
  fix(instrCounterClass r)
```

Oops, can't create objects!

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

newInstrCounter unit

→ let r = {x = ref 1, a = ref 0} in fix(instrCounterClass r)

→ fix(instrCounterClass *ivars*)

→ fix(λ self : InstrCounter .

 let super = setCounterClass *ivars* self in *imethods*)

→ let super = setCounterClass *ivars* (fix *f*) in *imethods*

→ let super = (λself: SetCounter . *smethods*) (fix *f*)
 in *imethods*

→ let super = (λself: SetCounter . *smethods*)

 (let super = setCounterClass *ivars* (fix *f*) in *imethods*)

 in *imethods*

→ ...

Possible Fixes

- suspend evaluation of self using **thunks**
- use call-by-name at critical points
- build loops using refs
- abandon lambda-calculus and create a new specialized calculus for OO

Thunks

Wrap a term in a lambda abstraction:

$$t : T$$

is replaced by

$$t' = \lambda _ : \text{Unit} . t : \text{Unit} \rightarrow T$$

where we don't want it evaluated, and by

$$t' \text{ unit}$$

where we do need its value.

Open Recursion with Thunks

```
setCounterClass =  
  λ r : CounterRep .  
    λ self : 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) ->  
    (Unit -> SetCounter)  
newSetCounter =  
  λ _ : Unit . let r = {x = ref 1} in fix (setCounterClass r) unit
```

Inheritance with Thunks

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

Replacing fix with ref

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

```
dummySetCounter: SetCounter =  
  {get = λ _ : Unit . 0,  
   set = λ i : Nat . unit,  
   inc = λ _ : Unit . unit}
```

```
newSetCounter : Unit -> SetCounter =  
  λ _ : Unit . let r = {x = ref 1} in  
    let cAux = ref dummySetCounter in  
    (cAux := (setCounterClass r cAux); !cAux)
```

Inheritance with ref

```
instrCounterClass =  
  λ r : InstrCounterRep . λ self : Ref InstrCounter .  
    let super = setCounterClass r self in  
      {get = super.get,  
       set = λ i : Nat. (r.a := succ(!(r.a)); super.set i),  
       inc = super.inc,  
       accesses = λ _ : Unit. !(r.a)}  
> instrCounterClass : InstrCounterRep -> (Ref InstrCounter)  
   -> InstrCounter
```

Type error: Ref InstrCounter </: Ref SetCounter

Replacing Ref with Source

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

Replacing Ref with Source, 2

```
instrCounterClass =  
  λ r : InstrCounterRep . λ self : Source InstrCounter .  
    let super = setCounterClass r self in  
      {get = super.get,  
       set = λ i : Nat. (r.a := succ(!(r.a)); super.set i),  
       inc = super.inc,  
       accesses = λ _ : Unit. !(r.a)}  
> instrCounterClass : InstrCounterRep -> (Source InstrCounter)  
   -> InstrCounter
```

Type checks: `Source InstrCounter <: Source SetCounter`

Replacing fix with ref

```
dummyInstrCounter: InstrCounter =  
  {get = λ _ : Unit . 0,  
   set = λ i : Nat . unit,  
   inc = λ _ : Unit . unit,  
   accesses = λ _ : Unit . 0}
```

```
newInstrCounter : Unit -> InstrCounter =  
  λ _ : Unit . let r = {x = ref 1, a = ref 0} in  
    let cAux = ref dummyInstrCounter in  
      (cAux := (instrCounterClass r cAux); !cAux)
```