Chapter 18: Case Study: Imperative Objects

What Is Object-Oriented Programming?
Objects/Class
Implementation
Review
Preliminary: syntax, operational semantics

Untyped lambda calculus

Simply typed lambda calculus

Simple extension: tuples/records, sums, lists

Subtyping

Universal type: system F

Case Study

Typing

ADT

Recursive Types
What is Object-Oriented Programming

- **Multiple representations**
  - Object (instances)

- **Encapsulation**
  - Internal representation/implementation is hidden

- **Subtyping**
  - Object interface (code reusing)

- **Inheritance**
  - Class, subclass, superclass

- **Open recursion.**
  - Self (this)

This chapter: lambda-calculus with subtyping, records, and references can model all these features!
Object

• object = internal state + set of methods

```
c = let \textbf{x} = ref 1 in
   \{get = \lambda _: \text{Unit. } !x, \\
   inc = \lambda _: \text{Unit. } x := \text{succ}(!x)\};
```
Object

- \( \text{object} = \text{internal state} + \text{set of operations} \)

\[
\begin{align*}
    c &= \text{let } x = \text{ref } 1 \text{ in} \\
    &\{ \text{get} = \lambda_\text{Unit}. \ x, \\
    &\quad \text{inc} = \lambda_\text{Unit}. \ x := \text{succ}(!x) \}; \\
    \Rightarrow c : \{ \text{get: Unit→Nat, inc: Unit→Unit} \}
\end{align*}
\]

\[
\text{Counter} = \{ \text{get : Unit→Nat, inc : Unit→Unit} \}
\]
Object

- **object invocation**

```plaintext
  c.inc unit;
  ▶ unit : Unit
  c.get unit;
  ▶ 2 : Nat
      (c.inc unit; c.inc unit; c.get unit);
  ▶ 4 : Nat
```
Object Generator

- A function that creates and returns a new counter every time it is called.

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

• newCounter : Unit -> Counter
```

Exercise: Can you define inc3 c to apply inc of a counter c three times?
Subtyping

- Permit objects of many shapes to be manipulated by the same client code.

```csharp
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
```

newResetCounter unit <: newCounter unit
Grouping Instance Variables

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

> c : Counter
```
Simple Classes

- Describing the common functionality in one place

Abstract the methods with respect to the instance variables

```
counterClass =
  λr: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;
```
Subclass

• The method bodies from one class can be reused to define new classes

```haskell
resetCounterClass =
  \r:CounterRep.
    let super = counterClass r in
      {get  = super.get,
       inc  = super.inc,
       reset = \_:Unit. r.x:=1};

  resetCounterClass : CounterRep → ResetCounter
```

```haskell
newResetCounter =
  \_:Unit. let r = \{x=ref 1\} in resetCounterClass r;

  newResetCounter : Unit → → ResetCounter
```
18.6.1 **Exercise [Recommended, ★★]:** Write a subclass of `resetCounterClass` with an additional method `dec` that subtracts one from the current value stored in the counter. Use the `fullref` checker to test your new class.
Adding Instance Variables in Subclasses

- How to define a class of “backup counters” whose reset method resets their state to whatever value it has when we last called the method backup, instead of resetting it to a constant value?

BackupCounter = {get: Unit→Nat, inc: Unit→Unit,
                  reset: Unit→Unit, backup: Unit→Unit};

BackupCounterRep = {x: Ref Nat, b: Ref Nat};
backupCounterClass =
    \( r : \text{BackupCounterRep} \).
    \( \text{let super = resetCounterClass } r \text{ in} \)
    \{ get = super.get,
      inc = super.inc,
      reset = \text{?},
      backup = \text{?} \}

| backupCounterClass : BackupCounterRep → BackupCounter |
backupCounterClass =
    \lambda 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
Calling Superclass Methods

- Extend the superclass’s behavior with something extra

```haskell
funnyBackupCounterClass =
  λr: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

• Allowing the methods of classes to refer to each other via `self`

```haskell
setCounterClass =
  λr:CounterRep.
    fix
      (λself: SetCounter.
        {get = λ_:Unit. !r.x),
         set = λi:Nat. r.x:=i,
         inc = λ_:Unit. self.set (succ (self.get unit)))
     );
>
setCounterClass : CounterRep → SetCounter
```

```haskell
newSetCounter =
  λ_:Unit. let r = {x=ref 1} in
    setCounterClass r;
>
newSetCounter : Unit → SetCounter
```
Open Recursion (Late Binding of Self)

“fix” is moved from class definition to object creation

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

  setCounterClass : CounterRep → SetCounter → SetCounter
```

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

  newSetCounter : Unit → SetCounter
```
• Advantage: allowing a superclass to call a method of a subclass

Example: building a subclass of our set-counters that keeps track of how many times the set method has been called:

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

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

\triangleright instrCounterClass : InstrCounterRep \to InstrCounter \to InstrCounter
```
Open Recursion and Evaluation Order

• Problem with `instrCounterClass`: we cannot use it to build instances!

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

newInstrCounter : Unit \rightarrow InstrCounter
```

```
ic = newInstrCounter unit;
```

• Solution: delay the evaluation of self

```plaintext
setCounterClass = 
  λr:CounterRep. 
  λself: Unit→SetCounter. 
      λ_:Unit. 
        \{get = λ_:Unit. !(r.x), 
          set = λi: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;

  \newSetCounter : Unit → SetCounter
```
instrCounterClass =
\lambda r: \text{InstrCounterRep}.
\lambda \text{self: Unit} - \text{InstrCounter}.
  \lambda _: \text{Unit}.
  \text{let super = setCounterClass r self unit in}
  \{get = super.get,
   set = \lambda i: \text{Nat}. (r.a:=\text{succ}(!r.a)); super.set i),
   inc = super.inc,
   accesses = \lambda _: \text{Unit}. !(r.a)\};

\triangleright \text{instrCounterClass : InstrCounterRep → (Unit} - \text{InstrCounter}) → \text{Unit} → \text{InstrCounter}

newInstrCounter =
\lambda _: \text{Unit}. \text{let r = \{x=ref 1, a=ref 0\} in}
  \text{fix (instrCounterClass r) unit;}

\triangleright \text{newInstrCounter : Unit → InstrCounter}
More Efficient Implementation

All the “delaying” we added has an unfortunate side effect:

Instead of computing the “method table” just once, when an object is created, we will now re-compute it every time we invoke a method!

Section 18.12 in the book shows how this can be repaired by using references instead of fix to “tie the knot” in the method table.