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

Universal type: system F

Subtyping

ADT

Typing

Recursive Types

Reference

Case Study

FJ
What is Object-Oriented Programming

• Multiple representations
  – Object (instances)

• Encapsulation
  – Internal representation/implementation is hidden

• Subtyping
  – Object interface

• 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 operations

\[
c = \text{let } x = \text{ref } 1 \text{ in }
\{
\text{get} = \lambda_\text{Unit}. !x,
\text{inc} = \lambda_\text{Unit}. x := \text{succ}(!x)\};
\]
Object

- object = internal state + set of operations

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

Counter = \{get : Unit \to \text{Nat}, inc : Unit \to \text{Unit}\}
Object

• object invocation

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

newResetCounter : Unit → ResetCounter
```

newResetCounter unit <: newCounter unit
Grouping Instance Variables

\[
c = \text{let } r = \{ x=\text{ref } 1 \} \text{ in }
\{ 
\text{get} = \lambda_\text{Unit.} \,! (r.x), \\
\text{inc} = \lambda_\text{Unit.} \ r.x := \text{succ}(! (r.x)) 
\};
\]

\[\text{c : Counter}\]
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

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

• 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, \text{backup: Unit→Unit}\};

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

> backupCounterClass : BackupCounterRep → BackupCounter
backupCounterClass =
    λr:BackupCounterRep.
    let super = resetCounterClass r in
    {get = super.get,
     inc = super.inc,
     reset = λ_:Unit. r.x:=!(r.b),
     backup = λ_:Unit. r.b:=!(r.x)};

* backupCounterClass : BackupCounterRep → BackupCounter
Calling Superclass Methods

• Extend the superclass's behavior with something extra

```haskell
funnyBackupCounterClass =
  \r:\BackupCounterRep.
  let super = backupCounterClass r in
  {get = super.get,
   inc = \_:_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

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

```haskell
setCounterClass =
    λr:CounterRep.
        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 → SetCounter

newSetCounter =
    λ_: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 =
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
```
Open Recursion and Evaluation Order

• Problem with \texttt{instrCounterClass}: we cannot use it to build instances!

\begin{verbatim}
newInstrCounter =
\lambda_#:Unit. \text{let } r = \{x=\text{ref } 1, a=\text{ref } 0\} \text{ in fix (instrCounterClass } r);\
\text{newInstrCounter : Unit } \rightarrow \text{ InstrCounter}
\end{verbatim}

\texttt{ic = newInstrCounter unit;}

\text{Object generator}

\text{Its evaluation diverges}

\text{WHY?}
• Solution: delay the evaluation of self

```plaintext
setCounterClass =
  \(r: \text{CounterRep} \rightarrow \text{Unit} \rightarrow \text{SetCounter} \rightarrow \text{SetCounter}) \rightarrow \text{Unit} \rightarrow \text{SetCounter} \rightarrow \text{SetCounter}
```

```plaintext
newSetCounter =
  \(\text{fix (setCounterClass r)} \text{ unit;})
```

• newSetCounter : Unit → SetCounter
Now the following computation will not diverge!

\[ \text{WHY?} \]

```haskell
instrCounterClass =
  \( r : \text{InstrCounterRep} \).
  \( \text{self} : \text{Unit} \rightarrow \text{InstrCounter} \).
  \( \text{Unit} \rightarrow \text{InstrCounter} \).

let super = setCounterClass \( r \) \( \text{self} \) \( \text{unit} \) in
  \{get = super.get,
   set = \( i : \text{Nat} \). (r.a:=succ(!(r.a)); super.set i),
   inc = super.inc,
   accesses = \( _ : \text{Unit} \). !(r.a)\};

\[ \text{instrCounterClass : InstrCounterRep} \rightarrow \]
\[ (\text{Unit} \rightarrow \text{InstrCounter}) \rightarrow \text{Unit} \rightarrow \text{InstrCounter} \]

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

\[ \text{newInstrCounter : Unit} \rightarrow \text{InstrCounter} \]

Now the following computation will not diverge!

\[ \text{WHY?} \]

\[ \text{ic} = \text{newInstrCounter} \text{ unit}; \]
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.