编程语言的设计原理
Design Principles of Programming Languages

Haiyan Zhao, Di Wang
赵海燕，王迪

Peking University, Spring Term 2023

Chap 18: Case Study: Imperative Objects

Embedding or Formalizing

What is Object-Oriented Programming?

Object / Class

Implementation

Review

Functional Programming

- Lambda-calculus
- Records
- General recursion
- Mutable references
- Subtyping

What about Other **Programming Paradigms**?

- Imperative programming
- Object-oriented programming
- Logic programming

Two Approaches to Defining a Language

Embedding in Lambda-Calculus

- Use lambda-calculus to encode programming idioms
- Can be thought as “syntax sugars”
- This chapter: **use lambda-calculus to approximate object-oriented programming**

Formalizing from Scratch

- Axiomatize the syntax, evaluation, and typing
- Follow the methodology in this course
- Next chapter: **formalize a subset of Java from scratch**

Embedding

Embedding Imperative Programming

WIKIPEDIA: “Imperative programming uses statements to change a program’s state.”

Remark

Mutable references model **state changes** in lambda-calculus.

```
int a = 1;        ⟹    let a = ref 1 in
a = a + 1;        ⟹    a := !a + 1;
return a;         ⟹    !a
```

Question

What about loops?

```
while (i < n) {
  int c = a + b;
  a = b; b = c; i = i + 1;
}
```

Embedding Imperative Programming

Remark

Recall **general recursion** via **fix** operator with $\mathbf{fix}\ f \equiv f\ (\mathbf{fix}\ f)$.

Evaluation and Typing Rules of fix

$$\frac{}{\mathbf{fix}\ (\lambda x{:}T_1.t_2) \longrightarrow [x \mapsto (\mathbf{fix}\ (\lambda x{:}T_1.t_2))]t_2}\ \textsc{E-FixBeta}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathbf{fix}\ t_1 \longrightarrow \mathbf{fix}\ t_1'}\ \textsc{E-Fix}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_1}{\Gamma \vdash \mathbf{fix}\ t_1 : T_1}\ \textsc{T-Fix}$$

Question

How to embed loops in lambda-calculus using general recursion?

Embedding Imperative Programming

```
while (i < n) {
  int c = a + b;
  a = b; b = c; i = i + 1;
}
```

$\Longrightarrow$

```
loop_gen =
  λ loop:(Unit→Unit). λ_:Unit.
    if !i < !n then
      let c = ref (!a + !b) in
      ( a := !b ; b := !c ; i := !i + 1 ;
        loop unit )
    else
      unit;
▶ loop_gen : (Unit→Unit)→Unit→Unit;
loop = fix loop_gen;
▶ loop : Unit→Unit;
loop unit;
▶ unit : Unit;
```

Summary

Lambda-calculus with mutable references and general recursion can encode imperative programming.

What is Object-Oriented Programming?

Object-Oriented Programming (OOP)

Wikipedia: "OOP is based on objects, which can contain data (as fields) and code (as methods)."

Example (Points in the Plane)

Consider implementing points as objects.

- **Data**: the representations of the point, e.g., cartesian form (x, y), polar form (r, θ), etc.
- **Code**: the operations for the point, e.g., its distance from the origin, its belonging quadrant, etc.

A set of operations (i.e., the **interface**) can be implemented differently based on the representations, e.g.:

$$\mathrm{dist}_{\mathrm{cart}}(x, y) \stackrel{\text{def}}{=} \sqrt{x^2 + y^2}$$

$$\mathrm{dist}_{\mathrm{pol}}(r, \theta) \stackrel{\text{def}}{=} r$$

Principle (I)

Multiple representations: Same interface can have different implementations.

Object-Oriented Programming (OOP)

Example (Points in the Plane)

A point's internal data should be **hidden** from outside.
Let us implement a function that checks whether a point lies in the unit circle.

$$\text{is\_in\_unit\_circle}(p) \overset{\text{def}}{=} (\text{dist}(p) < 1)$$

The function uses the dist method from the interface of points.
Thus, it works for **both** the cartesian form **and** the polar form.

Principle (II)

Encapsulation: Internal representation is hidden.

Embedding Objects in Lambda-Calculus

Objects

Remark

Recall that "object = internal data + interface methods."
We use **mutable references** to encode data and **records** to organize interface.

Example (Counters)

A counter object provides two methods:

- `get`: return the current counter value.
- `inc`: increment the counter.

```
c = let x = ref 1 in
        {get = λ_:Unit. !x,
         inc = λ_:Unit. x ≔ succ(!x)};
▶ c : {get:Unit→Nat, inc:Unit→Unit}
```

Objects

Example (Counters)

Invoke a method of an object = extract a field of its interface record and apply.

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

For convenience, let us define `Counter = {get:Unit→Nat, inc:Unit→Unit}`.

Question (In-Class Exercise)

Can you define `inc3 : Counter→Unit` that increments a counter three times?

Objects

Question

Can we define `newCounter` that generates a new counter? What should be its type?

```
newCounter =
  λ_:Unit. let x = ref 1 in
                 {get = λ_:Unit. !x,
                  inc = λ_:Unit. x := succ(!x)};
▶ newCounter : Unit→Counter
```

Question

Can we change the internal representation of the counters?

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

Object-Oriented Programming (OOP)

Summary

OOP principles so far:

1. **Multiple representations**: same interface can have different implementations.
2. **Encapsulation**: internal representation is hidden.

Question

Is that all?

What is Object-Oriented Programming? (cont.)

Code Reusing

Remark

OOP is good at **code reusing**: objects of different representations can be manipulated by the same code.

```
c = let x = ref 1 in
      {get = λ_:Unit. !x,
       inc = λ_:Unit. x := succ(!x)};
▶ c : Counter
inc3 c;
▶ unit : Unit
```

```
c = let r = {x=ref 1} in
      {get = λ_:Unit. !(r.x),
       inc = λ_:Unit. r.x := succ(!(r.x))};
▶ c : Counter
inc3 c;
▶ unit : Unit
```

Question

Given a function `inc3 : Counter→Unit`, can it be applied to values of other types?

Remark

We can use **subtyping**, i.e., if `d : T` for some `T <: Counter`, the term `inc3 d` is well-typed.

Subtyping

Principle (III)

Subtyping: Object-interface subtyping enables cross-interface code reusing.

Example (Counters)

Consider counters that can be reset:

```
ResetCounter = {get:Unit→Nat, inc:Unit→Unit, reset:Unit→Unit};
newResetCounter =
  λ_:Unit. let r = {x=ref 1} in
                {get   = λ_:Unit. !(r.x),
                 inc   = λ_:Unit. r.x := succ(!(r.x)),
                 reset = λ_:Unit. r.x := 1};
▶ newResetCounter : Unit→ResetCounter
```

Because `ResetCounter <: Counter`, we can apply `inc3` to reset-counters:

```
let d = newResetCounter unit in (inc3 d; d.reset unit; inc3 d; d.get unit);
▶ 4 : Nat
```

Code Reusing (cont.)

Question

The definitions of `newCounter` and `newResetCounter` are almost identical.
Can we describe the common functionality in one place?

Principle

A type = a set of **classes**, each with a distinct internal representation.
Recall that "the type of points = the class with cartesian form + the class with polar form."

Example (Counters)

```
CounterRep = {x : Ref Nat};
counterClass =
  λ r:CounterRep.
     {get = λ_:Unit. !(r.x),
      inc = λ_:Unit. r.x := succ(!(r.x))};
▶ counterClass : CounterRep→Counter
```

Inheritance

Example (Counters)

We can reuse methods from `counterClass` to define a new class `resetCounterClass`:

```
resetCounterClass =
  λ r:CounterRep.
    let super = counterClass r in
      {get   = super.get,
       inc   = super.inc,
       reset = λ_:Unit. r.x := 1};
▶ resetCounterClass : CounterRep→ResetCounter
```

In other words, `resetCounterClass` **inherits** `get` and `inc` **from** `counterClass`.

Principle (IV)

Inheritance: classes provide a mechanism to organize inheritance-based code reusing.

In-Class Exercise

Question (Exercise 18.6.1)

Write a subclass of `resetCounterClass` with an additional method `dec` that subtracts one from the current value stored in the counter.

You may test your new class using the `fullref` checker.

Adding Instance Variables

Question

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}
```

We need an extra instance variable to store the backed-up value:

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

```
backupCounterClass =
  λ r: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

Question

When defining a class, can we extend its superclass's behavior with something extra?

```
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

Question

Can we allow the methods of a class to refer to each other?
Suppose that we want to implement counters with a `set` method:

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

And we want to implement `inc` in terms of `get` and `set`.

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

Question

How to resolve such a mutually recursive record of functions?

Classes with Self

Remark

Recall **general recursion** via **fix** operator with **fix** $f \equiv f\ (\textbf{fix}\ f)$.

```
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
```

Object-Oriented Programming (OOP)

Summary

OOP principles so far:

I **Multiple representations**: same interface can have different implementations.
II **Encapsulation**: internal representation is hidden.
III **Subtyping**: Object-interface subtyping enables cross-interface code reusing.
IV **Inheritance**: classes provide a mechanism to organize inheritance-based code reusing.

Question

Is that all?

What is Object-Oriented Programming? (cont. again)

Dynamic Dispatch

Example (Counters)

We sometimes want to allow the methods of a **superclass** to call the methods of a **subclass**.

```
InstrCounter = {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit, accesses:Unit→Nat};
InstrCounterRep = {x : Ref Nat, a : Ref Nat};
instrCounterClass =
  λ r:InstrCounterRep.
     fix
       (λ self:InstrCounter.
          let super = setCounterClass r 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
```

However, the `inc` method from the superclass will **not** call the `set` method of the subclass.

Late Binding of Self

Principle (V)

Open recursion: `self` gets bound during object creation instead of class definition.

Example (Counters)

In the definition of `setCounterClass`, we make `self` a parameter:

```
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
```

Late Binding of Self

Example (Counters)

```
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

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

Does it really work?

```
    newInstrCounter unit
→*  let r = {x=ref 1, a= ref 0} in fix (instrCounterClass r)
→*  fix (instrCounterClass <ifields>)
→*  fix (λself:InstrCounter. let super = setCounterClass <ifields> self in <imethods>)
→*  let super = setCounterClass <ifields> (fix <f>) in <imethods>
→*  let super = (λself:SetCounter. <smethods>) (fix <f>) in <imethods>
→*  let super = (λself:SetCounter. <smethods>)
                   (let super = setCounterClass <ifields> (fix <f>) in <imethods>)
    in <imethods>
→*  ...
```

Problem

In the **call-by-value** evaluation order, the derivation above will infinitely unroll (**fix** <f>).

Solution

Use dummy lambda abstractions to control the evaluation order.

Late Binding of Self, Correctly

Example (Counters)

```
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;
▶ newSetCounter : Unit→SetCounter
```

Late Binding of Self, Correctly

Example (Counters)

```
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;
▶ newInstrCounter : Unit→InstrCounter
```

Object-Oriented Programming (OOP)

Summary

OOP principles:

I. **Multiple representations**: same interface can have different implementations.
II. **Encapsulation**: internal representation is hidden.
III. **Subtyping**: Object-interface subtyping enables cross-interface reusing.
IV. **Inheritance**: classes provide a mechanism to organize inheritance-based code reusing.
V. **Open recursion**: `self` gets bound during object creation instead of class definition.

Aside (Efficiency)

Instead of computing the "method table" just once when an object is created, we will **re-compute it every time** we invoke a method!
Section 18.12 in the book shows how this can be repaired by using **mutable references** instead of **`fix`** to "tie the knot" in the method table.