Imperative Objects

Advanced Compiler Construction and Program Analysis

Lecture 8

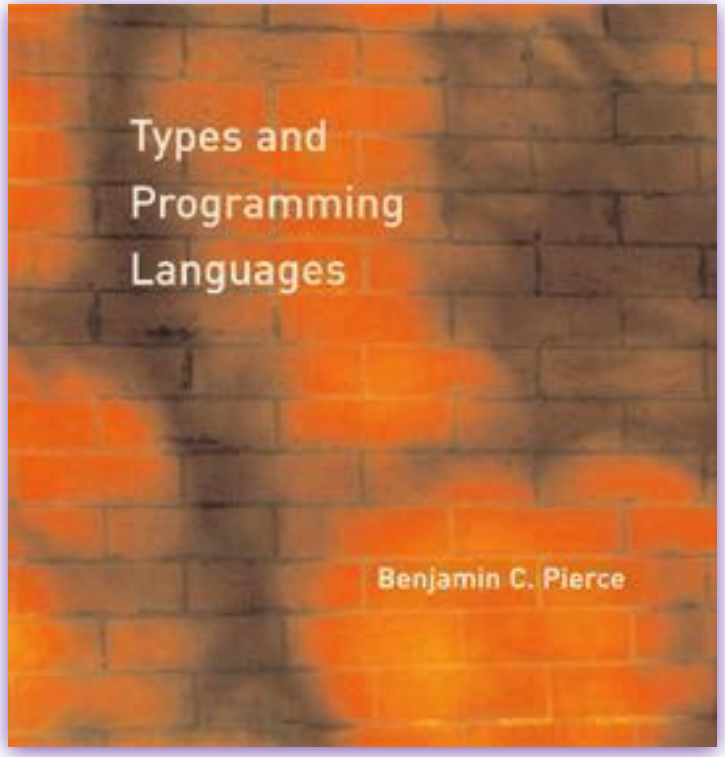
Innopolis University, Spring 2022

The topics of this lecture are covered in detail in…

Benjamin C. Pierce.

Types and Programming Languages

MIT Press 2002



What is OOP?

1. Multiple representations
2. Encapsulation
3. Subtyping
4. Inheritance
5. Open recursion

Simple Objects

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

Counter = { get : Unit → Nat, inc : Unit → Unit }

let inc3 =
  λc:Counter. (c.inc unit; c.inc unit; c.get unit)
```

Object Generators

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

newCounter : Unit → Counter
```

Subtyping of Objects

```
Counter =
    { get : Unit → Nat
    , inc : Unit → Unit }

ResetCounter =
    { get : Unit → Nat
    , inc : Unit → Unit
    , reset : Unit → Unit }

ResetCounter <: Counter
```

Grouping representation variables

```
CounterRep = { x : Ref Nat }


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

Simple Classes

Classes in modern languages are complicated, we will focus on the basics: **instantiating** and **extending** classes.

```
let counterClass : CounterRep → Counter =
  λr:CounterRep.
    { get = λ_:Unit. !(rep.x)
    , inc = λ_:Unit. rep.x := succ !(rep.x) } in
let newCounter : Unit → Counter =
  λ_:Unit.
    let rep = { x = ref 1 } in
      counterClass rep
```

Inheritance: extending a class

```
let resetCounterClass : CounterRep → ResetCounter =
  λrep:CounterRep.
    let super = counterClass rep in
      { get = super.get
      , inc = super.inc
      , reset = λ_:Unit. rep.x := 0 }
  in
let newResetCounter : Unit → ResetCounter =
  λ_:Unit.
    let rep = { x = ref 1 } in
      resetCounterClass rep
```

Inheritance: adding instance variables

Exercise 8.1. Implement `BackupCounter` class with methods `get`, `inc`, `reset` and `save`:

```
SaveCounter =
    { get   : Unit → Nat
    , inc   : Unit → Unit
    , reset : Unit → Unit
    , save  : Unit → Unit }
```

Inheritance: adding instance variables

```
SaveCounter =
    { get   : Unit → Nat
    , inc   : Unit → Unit
    , reset : Unit → Unit
    , save  : Unit → Unit }
```

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

```
let resetCounterClass : CounterRep → ResetCounter =
  λrep:BackupCounterRep.
    let super = resetCounterClass rep in
      { get = super.get
      , inc = super.inc
      , reset = λ_:Unit. rep.x := rep.backup
      , save = λ_:Unit. rep.backup := rep.x }
```

Classes with Self

Exercise 8.2. Implement `SetCounter` class with methods get, set, and inc:

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

Open recursion: logging counter

Exercise 8.3. Implement `LogCounter` as a subclass of `SetCounter` that it logs the number of accesses to `get`:

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

Summary

- ❏ Simple objects, subtyping
- ❏ Simple classes, inheritance
- ❏ Classes with Self
- ❏ Open recursion

See you next time!