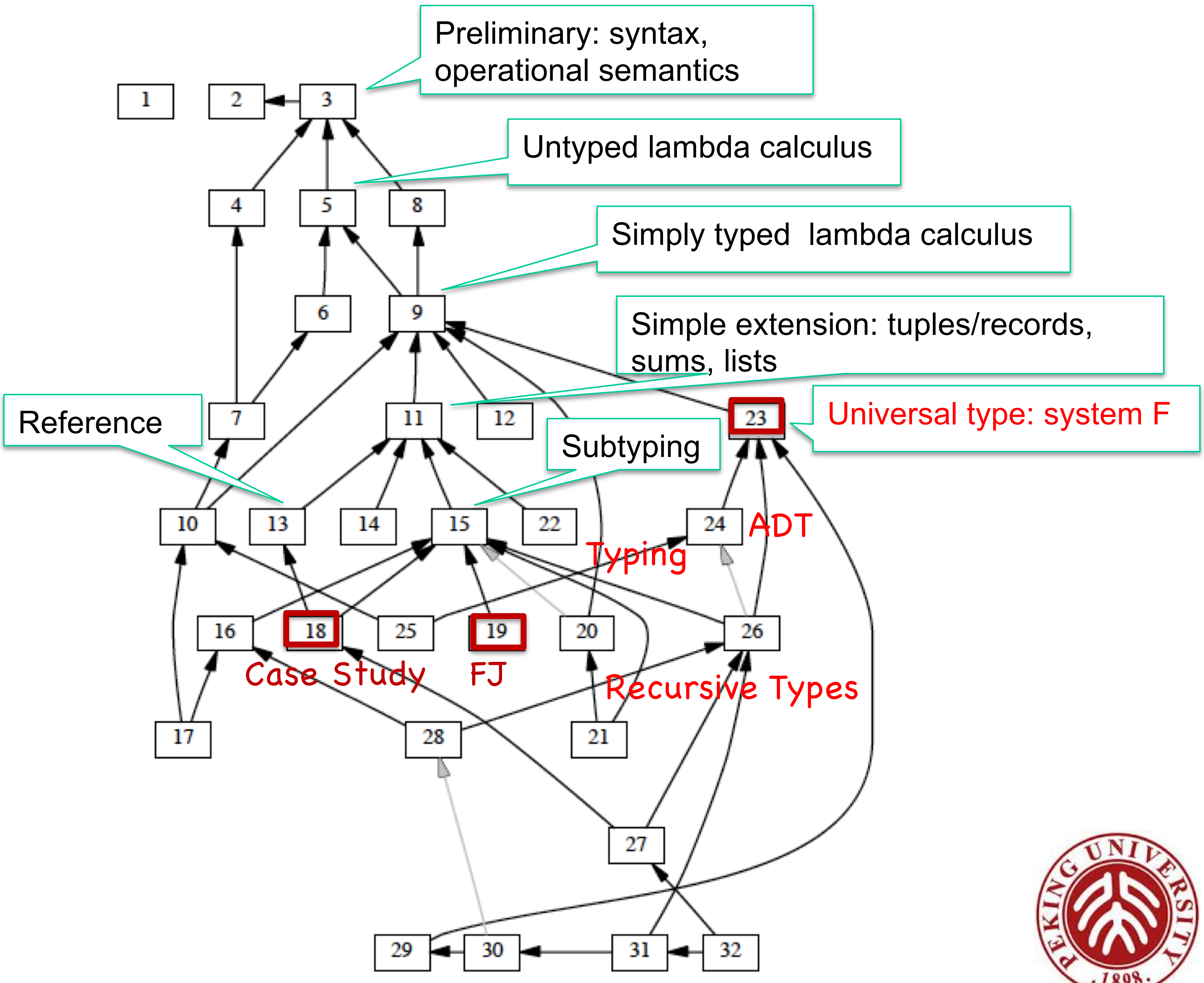
Chapter 18: Case Study: Imperative Objects

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
Reference
Subtyping
ADT
Typing
Case Study
FJ
Recursive Types
1
2
3
4
5
8
6
9
7
11
12
23
10
13
14
15
22
24
16
18
25
19
20
26
17
28
21
27
29
30
31
32

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

Object

- object = internal state + set of operations

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

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

Object

- object invocation

```
  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.

```
newCounter =
  λ_:Unit. let x = ref 1 in
             {get = λ_:Unit. !x,
              inc = λ_: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.

```
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

Allows a group of variables

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

```
resetCounterClass =
  λr: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
```

Exercise (at class)

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, backup: Unit→Unit};
```

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

```
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

```
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

```
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 instrCounterClass: we cannot use it to build instances!

Object generator

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

▸ newInstrCounter : Unit → InstrCounter
```

```
ic = newInstrCounter unit;
```

Its evaluation diverges

WHY?

- Solution: delay the evaluation of self

```
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 =
  λr:InstrCounterRep.
  λself: Unit→InstrCounter.
    λ_:Unit.
      let super = setCounterClass r self unit 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 →
                        (Unit→InstrCounter) → Unit → InstrCounter
```

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

▸ newInstrCounter : Unit → InstrCounter
```

Now the following computation will not diverge! WHY?

```
ic = newInstrCounter 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.