Explaining OOP through Typed Lambda Calculus

Eduardo Bonelli

April 25, 2019

Foundations of OOPL

- What is OOP?
- Propose fundamental building blocks of what one considers to be an OOPL
- Model must be rigorous
  - Precise definitions
  - So we can compare them with others
  - So that we can prove properties about them
  - So that we can assess impact of changes

Two Paths

1. Translate to a “low-level” formalism that is well-understood
   - Eg. Lambda Calculus
2. Primitive formalization of representative notions
   - Eg. Featherweight Java

Translation to Lambda Calculus

We will use Lambda Calculus with

- functions
- records
- references
- recursion
- subtyping

In order to formalize:

- dynamic method dispatch
- state encapsulation
- inheritance
- this
- super

Example Object in Java

```
class Counter {
 protected int x = 1;
 int  get() { return x }
 void inc() { x++; }
}

void inc3(Counter c) {
 c.inc(); c.inc(); c.inc();
}

Counter c = new Counter();
inc3(c);
inc3(c);
c.get();
```

Same Object in Lambda Calculus

- Let

```
    Counter={get:Unit -> Nat, inc:Unit -> Unit}
```

- Then

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

- The notation

$$\texttt{e}\Longrightarrow\texttt{r}$$

should be read

"Evaluate e yielding r as result"

Objects

If we declare

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

$\Longrightarrow$

```
inc3: Counter -> Unit
```

Then

```
(inc3 c; inc3 c; c.get unit);
```

$\Longrightarrow$

7

New Object Generation

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

For example

```
nc = newCounter unit;
⟹
nc : Counter
```

Grouping Instance Variables

- Objects typically have multiple instance variables
- We can group them into records

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

The local variable `r` has type `CounterRep = {x: Ref Nat}`

Subtyping and Inheritance

```
class Counter {
 protected int x = 1;
 int  get() { return x; }
 void inc() { x++; }
}

class ResetCounter extends Counter {
 void reset() { x = 1; }
}

ResetCounter rc = new ResetCounter();
inc3(rc);
rc.reset();
inc3(rc);
rc.get();
```

Attempt to Encode in Lambda Calculus

```
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
```

Attempt to Encode in Lambda Calculus

```
rc = newResetCounter unit;
(inc3 rc; rc.reset unit; inc3 rc; rc.get unit);
```

$\Longrightarrow$

```
4
```

Summary

- Everything looks good

Java	Lambda Calculus
Class	Record type of the public interface
Class instantiation C	`newC:Unit -> C`

- We can also deal with multiple instance variables
- However `newResetCounter` does not make use of the extant implementation of the methods `get` and `inc`

```
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};
```

Towards an Encoding of Classes

- The inconvenience with the definition of `newResetCounter` is that it does not reuse the extant implementation of the methods `get` and `inc`
- What about this?

```
newResetCounterFromCounter =
  λc:Counter. let r = {x=ref 1} in
                     {get = c.get,
                      inc = c.inc,
                      reset = λ_:Unit. r.x:=1};
```

Towards an Encoding of Classes

- The inconvenience with the definition of `newResetCounter` is that it does not reuse the extant implementation of the methods `get` and `inc`
- What about this?

```
newResetCounterFromCounter =
  λc:Counter. let r = {x=ref 1} in
                      {get = c.get,
                       inc = c.inc,
                       reset = λ_:Unit. r.x:=1};
```

- Does not work: `reset` does not have access to the local variable `r` of the original counter

Classes

Should allow one to

1. Instantiate new objects
2. be extended with subclasses

Encoding Classes

- In order to avoid the above mentioned problem we separate the definition of methods...

```
counterClass =
  λr:CounterRep.
    {get = λ_:Unit. !(r.x),
     inc = λ_:Unit. r.x:=succ(!(r.x))};
⟹ counterClass : CounterRep -> Counter
```

- ...from the act of associating them to a particular set of instance variables

```
newCounter = λ_:Unit. let r = {x=ref 1} in
                          counterClass r;
⟹ newCounter : Unit -> Counter
```

Encoding Classes

More generally:

- A class `CClass` is represented as aa function

```
CClass : CRep -> C
```

- Type `CRep` is the representation of the state (i.e. instance variables)
- Type `C` is the type of the class itself (i.e. the type of the instances of `C`)

Subclassing

```
resetCounterClass =
  λr:CounterRep.
     let super = counterClass r in
        {get = super.get,
         inc = super.inc,
         reset = λ_:Unit. r.x:=1};
⟹ resetCounterClass : CounterRep -> ResetCounter
```

- Constructor resetCounterClass invokes the constructor counterClass
- Inheritance = nesting of local declarations

Subcliassing

```
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
```

- Constructor resetCounterClass invokes the constructor counterClass

Method Override and New Instance Variables

```
class Counter {
 protected int x = 1;
 int  get() { return x; }
 void inc() { x++; }
}

class ResetCounter extends Counter {
 void reset() { x = 1; }
}

class BackupCounter extends ResetCounter {
 protected int b = 1;
 void backup() { b = x; }
 void reset() { x = b; }
}
```

Adding Instance Variables

```
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 = λ_:Unit. r.x:=!(r.b),
        backup = λ_:Unit. r.b:=!(r.x)};
⟹
backupCounterClass : BackupCounterRep -> BackupCounter
```

Adding Instance Variables

- Note: `backupCounterClass` extends (with `backup`) and also overrides (with `reset`) the definition of `counterClass`

```
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
```

Adding Instance Variables

Subtyping is crucial

- `resetCounterClass : CounterRep -> ResetCounter`
- `CounterRep = {x:Ref Nat}`
- `BackupCounterRep = {x:Ref Nat, b:Ref Nat}`

`BackupCounterRep<:CounterRep`

```
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)};
```

Super

- Suppose we want a subclass of `BackupCounterClass`
- It should modify `inc` so that it first backs up the state before updating it

```
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 -> BackupCounte
```

Motivating “this”

- Suppose counters have methods `set`, `get` and `inc`

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

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

Motivating “this”

- Suppose counters have methods `set`, `get` and `inc`

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

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

- Behavior of `inc` could be specified via `get`/`set`

In Java

```
class SetCounter {
 protected int x = 1;
 int  get() { return x; }
 int  set(int i) { x=i; }
 void inc() { this.set( this.get()+1 ); }
}
```

"This" as a Fixed-Point – Motivation

Suppose we have an object instance of `SetCounter`

```
newSetCounter  = λ_:Unit. let r = {x=ref 1} in
                  setCounterClass r;
⟹ newCounter : Unit -> Counter
```

If `vr` is the value of `r`, then `newSetCounter ()` is the object:

```
{get = λ_:Unit. !(vr.x),
 set = λi:Nat. vr.x:=i,
 inc = λ_:Unit. vr.x:=succ(!(vr.x))}
```

Recall that we want to write `inc` in terms of `this`, just like in the Java code

“This” as a Fixed-Point – Motivation

In other words, from

```
{get = λ_:Unit. !(vr.x),
 set = λi:Nat. vr.x:=i,
 inc = λ_:Unit. vr.x:=succ(!(vr.x))}
```

we want to obtain:

```
{get = λ_:Unit. !(vr.x),
 set = λi:Nat. vr.x:=i,
 inc = λ_:Unit. this.set (succ (this.get unit))}
```

and provide meaning to `this` in the term above

```
this = {get = λ_:Unit. !(vr.x),
        set = λi:Nat. vr.x:=i,
        inc = λ_:Unit. this.set (succ (this.get unit))}
```

From an operational point of view, we can consider an unbounded number of unfoldings as the meaning:

```
this = {get = λ_:Unit. !(vr.x),
        set = λi:Nat. vr.x:=i,
        inc = λ_:Unit. {get = λ_:Unit. !(vr.x),
                         set = λi:Nat. vr.x:=i,
                         inc = λ_:Unit. ....set (succ
                            (....get unit))};.set
            (succ ({get = λ_:Unit. !(vr.x),
                    set = λi:Nat. vr.x:=i,
                    inc = λ_:Unit. ....set (succ (....get un
```

`this` as a Fixed Point

- Unbounded unfoldings = `fix`

```
fix
 (λthis:SetCounter.
     {get = λ_:Unit. !(vr.x),
      set = λi:Nat. vr.x:=i,
      inc = λ_:Unit. this.set (succ (this.get unit))}
```

- Small-step semantics of fix (from previous class)

$$\frac{}{fix(\lambda f{:}\sigma.M) \to M\{f{:=}fix(\lambda f{:}\sigma.M)\}}$$

Putting it All Together

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

newSetCounter =
 λ_:Unit. let r = {x=ref 1} in
          fix (setCounterClass r);
⟹ newSetCounter : Unit -> SetCounter
```

Example Continued

- We now want a new kind of counter
- It should be a subclass of `setCounter`
- Its instances should record the number of calls to `set`

```
InstrCounter = {get:Unit->Nat, set:Nat->Unit,
                inc:Unit->Unit, accesses:Unit-> Nat}

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

```
instrCounterClass =
  λr:InstrCounterRep.
    (λthis:InstrCounter.
       let super = setCounterClass r this 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
```

```
instrCounterClass =
  λr:InstrCounterRep.
    (λthis:InstrCounter.
       let super = setCounterClass r this in
      {get = super.get,
       set = λi:Nat. (r.a:=succ(!(r.a)); super.set i),
       inc = super.inc,
       accesses λ_:Unit. !(r.a)};
```

- `this` is passed on to the superclass as argument
- Hence, every reference to `this` in the superclass (i.e. `setCounterClass`) correctly referes to `InstrCounterClass`
- Eg. `inc` and `super` will call the `set` method of `instrCounterClass`, which in turn calls `set` of the superclass

```
instrCounterClass =
  λr:InstrCounterRep.
     (λthis:InstrCounter.
     let super = setCounterClass r this 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 -> I
```

- Subtyping is crucial, yet again, in the call to `setCounterClass`

A Problem with Object Creation

- What happens when we try to create an instance of the class `instrCounterClass`?

```
newInstrCounter unit
```

where

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

A Problem with Object Creation

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

$$
\begin{array}{ll}
 & \texttt{newInstrCounter unit} \\
\rightarrow & \text{let r} = \{\text{x} = \text{ref } 1, \text{ a} = \text{ref } 0\} \text{ in fix(instrCounterClass r)} \\
\rightarrow & \text{fix(instrCounterClass ivars)} \\
\rightarrow & \text{fix}(\lambda\text{self : InstrCounter .} \\
 & \quad \text{let super} = \text{setCounterClass ivars self in imethods}) \\
\rightarrow & \text{let super} = \underline{\text{setCounterClass ivars (fix f)}} \text{ in imethods} \\
\rightarrow & \text{let super} = (\lambda\text{self: SetCounter . smethods) (fix f) in imethods} \\
\rightarrow & \text{let super} = (\lambda\text{self: SetCounter . smethods)} \\
 & \qquad (\text{let super} = \underline{\text{setCounterClass ivars (fix f)}} \text{ in imethods}) \\
 & \quad \text{in imethods} \\
 & \ldots
\end{array}
$$

Solution

- Delay evaluation of `this` placing it under an abstraction
- Use call-by-name at certain critical points in the program
- Use references to records rather than fixed points for object representation
- Set aside LC and use different formalism to model OOP

Thunks

Place term M of type σ under an abstraction:

M : σ

is replaced with

M'=λ_:Unit.M: Unit -> σ

It is called with

M' unit

Thunks

```
setCounterClass =
  λr:CounterRep.
    λthis:Unit->SetCounter.
     λ_:Unit.
      {get = λ_:Unit. !(r.x),
       set = λi:Nat. r.x:=i,
       inc = λ_:Unit. (this 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;
```

Similar for instrCounterClass

```
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)};

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

Use Ref instead of fix for Objects

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

dummySetCounter: SetCounter =
       {get = λ_:Unit. 0,
        set = λi:Nat. unit,
        inc = λ_:Unit. unit}

newSetCounter : Unit -> SetCounter =
 λ_:Unit. let r = {x = ref 1} in
             let cAux = ref dummySetCounter in
                (cAux := (setCounterClass r cAux); !cAux)
```

Problem with Inheritance

```
instrCounterClass =
   λr:InstrCounterRep.
     λself:Ref 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 -> (Ref InstrCounte
                            -> InstrCounter
```

Type error: `Ref InstrCounter </: Ref SetCounter`

Replace Ref with Source (1/2)

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

Replace Ref with Source (2/2)

```
InstrCounterClass =
   λr:InstrCounterRep. λself:Source 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 ->
    (Source InstrCounter) -> InstrCounter
```

Type checks: `Source InstrCounter <: Source SetCounter`

Replace fix with ref

```
dummyInstrCounter: InstrCounter =
        {get = λ_:Unit. 0,
         set = λi:Nat. unit,
         inc = λ_:Unit. unit,
         accesses = λ_:Unit. 0}

newInstrCounter : Unit -> InstrCounter =
  λ_:Unit. let r = {x = ref 1, a = ref 0} in
               let cAux = ref dummyInstrCounter in
                (cAux := (instrCounterClass r cAux); !cAux)
```

Recall

```
Ref T <:  Source T
 Ref T <: Sink T
```