

inc =  $\lambda_:$ Unit. r.x:=(succ r.x) });

Bad style: The functionality of inc could be expressed in terms of the functionality of get and set.

Can we rewrite this class so that the get/set functionality appears just once?

Check: the type of the inner  $\lambda$ -abstraction is SetCounter $\rightarrow$ SetCounter, so the type of the fix expression is SetCounter.

This is just a definition of a set (record) of mutually recursive functions. (We saw something similar in the iseven/isodd example in 11.11.)

```
Note that the fixed point in setCounterClass =
     \lambda r: CounterRep.
       fix
          (\lambda self: SetCounter.
            \{get = \lambda_: Unit. ! (r.x), \}
             set = \lambdai:Nat. r.x:=i,
             inc = \lambda_:Unit. self.set (succ (self.get unit))});
  is "closed" - we "tie the knot" when we build the record.
  So this does not model the behavior of self (or this) in real OO
  languages.
CIS 500, 26 November
  Note that we have changed the types of classes from ....
  setCounterClass =
    \lambda r: CounterRep.
       fix
          (\lambda self: SetCounter.)
            {get = \lambda_{:}Unit. !(r.x),
             set = \lambdai:Nat. r.x:=i,
             inc = \lambda_{::Unit. self.set (succ (self.get unit))});
  \implies setCounterClass : CounterRep \rightarrow SetCounter
  ... to:
  setCounterClass =
    \lambda r: CounterRep.
        \lambdaself: SetCounter.
           {get = \lambda_{:}Unit. !(r.x),
            set = \lambdai:Nat. r.x:=i,
            inc = \lambda_{::Unit. self.set (succ(self.get unit))};
  \implies setCounterClass : CounterRep \rightarrow SetCounter \rightarrow SetCounter
```

```
CIS 500, 26 November
```

```
Using self
```

Let's continue the example by defining a new class of counter objects (a subclass of set-counters) that keeps a record of the number of times the set method has ever been called.

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

4

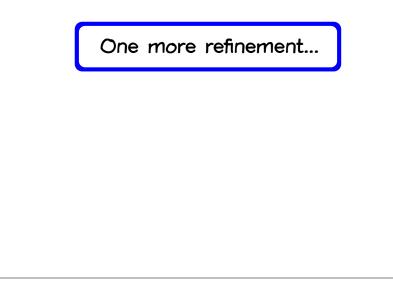
5

### instrCounterClass =

```
http://www.instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.com/instruction.c
```

CIS 500, 26 November

8



CIS 500, 26 November

A small fly in the ointment

The implementation we have given for instrumented counters is not very useful because calling the object creation function

newInstrCounter =

```
\lambda_:Unit. let r = {x=ref 1, a=ref 0} in
```

fix (instrCounterClass r);

will cause the evaluator to diverge!

Intuitively (see TAPL for details), the problem is the "unprotected" use of self in the call to setCounterClass in instrCounterClass:

### instrCounterClass =

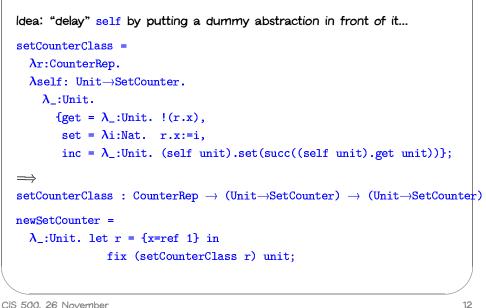
```
λr:InstrCounterRep.
    λself: InstrCounter.
```

```
let super = setCounterClass r self in
```

```
•••
```

To see why this diverges, consider a simpler example:  $ff = \lambda f: Nat \rightarrow Nat.$  let f' = f in  $\lambda n: Nat. 0$   $\implies ff : (Nat \rightarrow Nat) \rightarrow (Nat \rightarrow Nat)$ Now: fix ff  $\longrightarrow$  ff (fix ff)  $\longrightarrow let f' = (fix ff) in \lambda n: Nat. 0$   $\longrightarrow let f' = ff (fix ff) in \lambda n: Nat. 0$   $\longrightarrow uh oh...$  a

## One possible solution



CIS 500, 26 November

### Similarly:

instrCounterClass =  $\lambda r: InstrCounterRep.$  $\lambda$ self: Unit $\rightarrow$ InstrCounter.  $\lambda$  :Unit. let super = setCounterClass r self unit in {get = super.get, set =  $\lambda$ i:Nat. (r.a:=succ(!(r.a)); super.set i), inc = super.inc, accesses =  $\lambda_:$ Unit. !(r.a)};

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

CIS 500, 26 November

### Success

This works, in the sense that we can now instantiate instrCounterClass (without diverging!), and its instances behave in the way we intended.

## Success (?)

This works, in the sense that we can now instantiate instrCounterClass (without diverging!), and its instances behave in the way we intended.

However, 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 TAPL shows how this can be repaired by using references instead of fix to "tie the knot" in the method table.

13

## Multiple representations

All the objects we have built in this series of examples have type Counter. But their internal representations vary widely.

# Encapsulation

An object is a record of functions, which maintain common internal state via a shared reference to a record of mutable instance variables.

This state is inaccessible outside of the object because there is no way to name it. (Instance variables can only be named from inside the methods.)

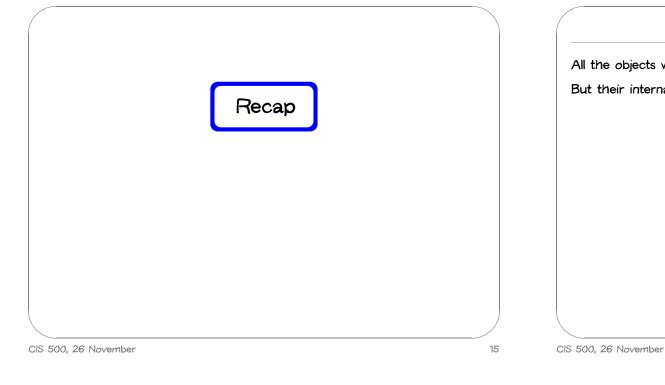
# Subtyping

Subtyping between object types is just ordinary subtyping between types of records of functions.

Functions like inc3 that expect Counter objects as parameters can (safely) be called with objects belonging to any subtype of Counter.

17

16



## Inheritance

Classes are data structures that can be both extended and instantiated.

We modeled inheritance by copying implementations of methods from superclasses to subclasses.

### Each class

- waits to be told a record r of instance variables and an object self (which should have the same interface and be based on the same record of instance variables)
- uses r and self to instantiate its superclass
- constructs a record of method implementations, copying some directly from super and implementing others in terms of self and super.
- The self parameter is "resolved" at object creation time using fix.

#### CIS 500, 26 November

19

## Additional exercise

Take all the examples from this lecture (and the previous one), and recode them in Java.

### [Not to be handed in - just for you to check your understanding.]

CIS 500, 26 November