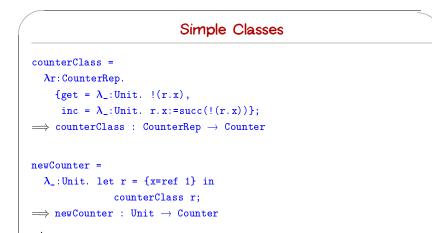




Counter = {get:Unit→Nat, inc:Unit→Unit}

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



Review

where...

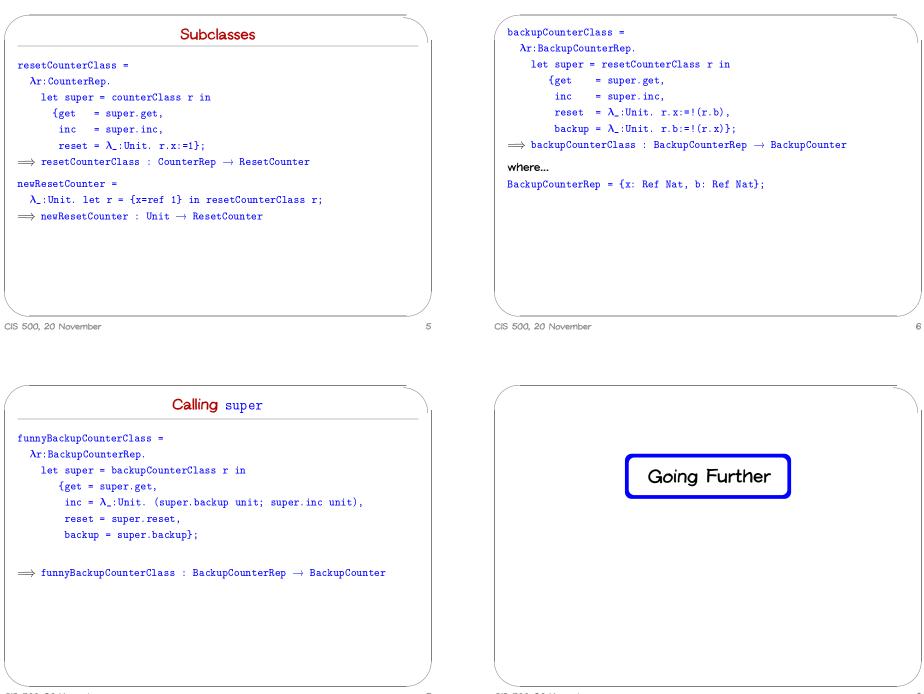
CIS 500, 20 November

CounterRep = {x: Ref Nat};

CIS 500, 20 November

CIS 500, 20 November

2



CIS 500, 20 November

7

CIS 500, 20 November

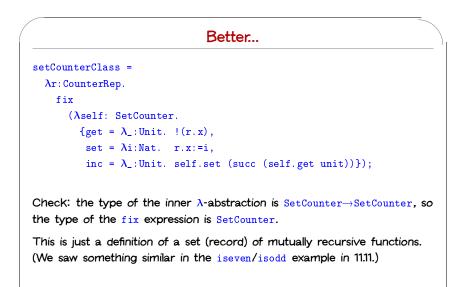
Calling between methods

Let's define a class of counters with set, get, and inc methods: SetCounter = {get:Unit→Nat, set:Nat→Unit, inc:Unit→Unit};

```
setCounterClass =
```

```
\begin{split} \lambda r: \text{CounterRep.} \\ & \{ \text{get} = \lambda_: \text{Unit.} | (r.x), \\ & \text{set} = \lambda i: \text{Nat.} r.x:=i, \\ & \text{inc} = \lambda_: \text{Unit.} r.x:=(\text{succ} r.x) \} ); \end{split}
```

CIS 500, 20 November



Calling between methods

Let's define a class of counters with set, get, and inc methods: SetCounter = {get:Unit \rightarrow Nat, set:Nat \rightarrow Unit, inc:Unit \rightarrow Unit}; setCounterClass = $\lambda r:CounterRep.$ {get = $\lambda_-:Unit. !(r.x),$ set = $\lambda i:Nat. r.x:=i,$ inc = $\lambda_-:Unit. r.x:=(succ r.x)$ }; Bad style: The functionality of inc could be expressed in terms of the

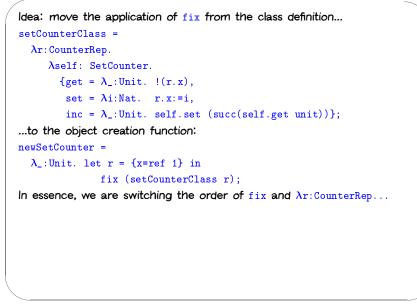
functionality of get and set. Can we rewrite this class so that the get/set functionality appears just

once?

CIS 500, 20 November

9-a

9



```
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, 20 November

12

CIS 500, 20 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};

```
CIS 500, 20 November
```

14

CIS 500, 20 November

<pre>instrCounterClass =</pre>
$\lambda r: \texttt{InstrCounterRep}$.
λ self: InstrCounter.
<pre>let super = setCounterClass r self in</pre>
<pre>{get = super.get,</pre>
set = λ i:Nat. (r.a:=succ(!(r.a)); super.set i),
<pre>inc = super.inc,</pre>
accesses = $\lambda_{:}:Unit. !(r.a)$;
\Rightarrow

 $\texttt{instrCounterClass} \ : \ \texttt{InstrCounterRep} \ \rightarrow \ \texttt{InstrCounter} \ \rightarrow \ \texttt{InstrCounter}$

Notes:

- the methods use both self (which is passed as a parameter) and super (which is constructed using self and the instance variables)
- the inc in super will call the set defined here, which calls the superclass set
- suptyping plays a crucial role (twice) in the call to setCounterClass

A small fly in the ointment The implementation we have given for instrumented counters is not very useful because calling the object creation function One more refinement... newInstrCounter = λ_{\perp} :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 = $\lambda r: InstrCounterRep.$ λ self: InstrCounter. let super = setCounterClass r self in CIS 500, 20 November 16 CIS 500, 20 November 17

```
To see why this diverges, consider a simpler example:

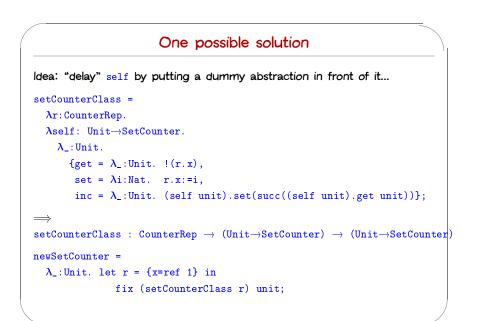
ff = \lambda f: Nat \rightarrow Nat.
let f' = f in
\lambda n: Nat. 0
\implies ff : (Nat \rightarrow Nat) \rightarrow (Nat \rightarrow Nat)
Now:

fix ff \rightarrow ff (fix ff)

\rightarrow let f' = (fix ff) in \lambda n: Nat. 0

\rightarrow let f' = ff (fix ff) in \lambda n: Nat. 0

\rightarrow uh oh...
```



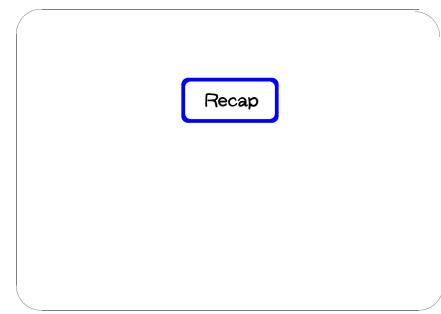
Similarly: Success instrCounterClass = This works, in the sense that we can now instantiate instrCounterClass $\lambda r: InstrCounterRep.$ (without diverging!), and its instances behave in the way we intended. λ self: Unit \rightarrow InstrCounter. λ_{\perp} :Unit. let super = setCounterClass r self unit in {get = super.get, set = λ i:Nat. (r.a:=succ(!(r.a)); super.set i), inc = super.inc, accesses = $\lambda_{::Unit. !(r.a)};$ newInstrCounter = λ_{\perp} :Unit. let r = {x=ref 1, a=ref 0} in fix (instrCounterClass r) unit; CIS 500, 20 November CIS 500, 20 November 20 21

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.



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.)

CIS 500, 20 November

23

CIS 500, 20 November

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.

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.

24