

Language Support for Object-oriented Programming

- ▶ Multiple representations of the same interface
- ▶ Encapsulation
- ▶ Subtyping
- ▶ Inheritance
- ▶ Open recursion (in a method, you can call other methods on “self” (“this”).

Objects

Here's an object that represents a counter. We use lexical scoping, references, and records.

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

```
c.inc unit;  
> unit : Unit
```

```
c.get unit;  
> 2 : Nat
```

Object Constructors

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

```
newCounter =
```

```
  λ_:Unit.
```

```
    let x = ref 1 in
```

```
      {
```

```
        get = λ_:Unit. !x,
```

```
        inc = λ_:Unit. x := succ !x
```

```
      };
```

```
> newCounter : Unit → Counter
```

```
newCounter unit;
```

```
> {get=λ ..., inc=λ ...} : Counter
```

Subtyping

- ▶ Suppose we also have some objects with the following type:

$$\text{ResetCounter} = \{ \text{get} : \text{Unit} \rightarrow \text{Nat}, \text{inc} : \text{Unit} \rightarrow \text{Unit}, \text{reset} : \text{Unit} \rightarrow \text{Unit} \}$$

- ▶ The subtype rule for record width allows an object of type `ResetCounter` in a place that expects a `Counter`.

$$\begin{aligned} \text{Counter} &= \{ \text{get} : \text{Unit} \rightarrow \text{Nat}, \text{inc} : \text{Unit} \rightarrow \text{Unit} \}; \\ \text{inc3} &= (\lambda c : \text{Counter}. c.\text{inc unit}; c.\text{inc unit}; c.\text{inc unit}); \end{aligned}$$
$$\begin{aligned} \text{newResetCounter} &= \\ &\lambda \text{init} : \text{Unit}. \mathbf{let} \ x = \text{ref init} \ \mathbf{in} \\ &\quad \{ \text{get} = \lambda _ : \text{Unit}. !x, \\ &\quad \quad \text{inc} = \lambda _ : \text{Unit}. x := \text{succ } !x \\ &\quad \quad \text{reset} = \lambda _ : \text{Unit}. x := \text{init} \}; \end{aligned}$$
$$\begin{aligned} \text{rc} &= \text{inc3} (\text{newResetCounter unit}); \\ > \text{rc} &: \text{ResetCounter} \\ \text{rc.get unit}; \\ > &4 : \text{Nat} \end{aligned}$$

Objects with Multiple Data Members

```
newResetCounter =  
  λini:Unit. let r = { x = ref ini, init = ini } in  
    {  
      get = λ_:Unit. !r.x,  
      inc = λ_:Unit. r.x := succ !r.x  
      reset = λ_:Unit. r.x := r.init  
    };
```

```
rc = newResetCounter 0;  
> rc : ResetCounter
```

Simple Classes

- ▶ The definition of `newCounter` and `newResetCounter` are rather similar.
- ▶ Wouldn't it be nice to reuse the code from `newCounter` when defining `newResetCounter`?

```
CounterRep = { x : Ref Nat }
```

```
counterClass =
```

```
  λr:CounterRep. {  
    get = λ_:Unit. !r.x,  
    inc = λ_:Unit. r.x := succ !r.x  
  };
```

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

Simple Classes

```
ResetCounterRep = { x : Ref Nat, init : Nat }
```

```
resetCounterClass =  
  λr:ResetCounterRep.  
    let super = counterClass r in  
      {  
        get = super.get,  
        inc = super.inc,  
        reset = λ_:Unit. r.x := r.init  
      };
```

```
newResetCounter =  
  λini:Nat. let r { x=ref ini, init = ini } in resetCounterClass r;
```

Mutually Recursive Methods via References

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

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

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

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

```
sc = newSetCounter unit;  
sc.inc unit;  
sc.get unit;
```

Method Overriding

```
InstrCounterRep = {x:Ref Nat, a:Ref Nat};  
InstrCounter = {get:Unit→ Nat, set:Nat→ Unit,  
               inc:Unit→ Unit, accesses:Unit→ Nat};  
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 };  
dummyInstrCounter =  
  { get = λ_:Unit. 0, set = λi:Nat. unit, inc = λ_:Unit. unit,  
    accesses = λ_:Unit. 0 };  
newInstrCounter = λ_ : Unit. let r = { x = ref 1, a = ref 0 } in  
  let cAux = ref dummyInstrCounter in  
    (cAux := (instrCounterClass r cAux); !cAux);  
ic = newInstrCounter unit;  
ic.inc unit; ic.inc unit; ic.accesses unit;
```

Why not Objects via Self-Application

- ▶ Why don't we do the normal thing of passing the object as the first parameter of each method?
- ▶ We would need **recursive** types for this:

$$\text{Counter} = \{ \text{get} : \text{Counter} \rightarrow \text{Nat}, \text{inc} : \text{Counter} \rightarrow \text{Unit} \};$$

- ▶ We'll study recursive types in a few weeks.