Chapter 18:
Case Study: Imperative Objects

Essence of Object-oriented programming

Objects / Objects Generators

Subtyping / Grouping Instance Variables

Simple Classes / Adding Instance Variables

Calling Superclass Methods / Classed with Self

Open Recursion through Self

Open Recursion and Evaluation Order

Change!!

We have focuses on developing tools for *defining and reasoning* about programming language features in the past 7 weeks .

Now it’s time to use these tools for something more ambitious.

Plan

1. Identify some characteristic "*core features*" of object-oriented programming
2. Develop *two different analysis* of these features:
    - 2.1 A *translation* into a lower-level language
    - 2.2 A *direct*, high-level formalization of a simple object-oriented language ("Featherweight Java")

The Translational Analysis

The first will be to show how many of the basic features of object-oriented languages

- *dynamic dispatch*
- *encapsulation of state*
- *inheritance*
- *late binding (this)*
- *super*

can be understood as *"derived forms"* in a lower-level language with a rich collection of primitive features:

- *(higher-order) functions*
- *records*
- *references*
- *recursion*
- *subtyping*

The Translational Analysis

For *simple objects and classes*, this translational analysis works very well.

When we come to *more complex features* (in particular, classes with this), it becomes less satisfactory

- the more direct treatment in the following chapter

Concepts

The Essence of Objects

What “is” object-oriented programming?

The Essence of Objects

What “is” object-oriented programming?

This question has been a subject of debate for decades. Such arguments are always *inconclusive* and *seldom very interesting*.

The Essence of Objects

What “is” object-oriented programming?

This question has been a subject of debate for decades. Such arguments are always inconclusive and seldom very interesting.

However, it is easy to identify some *core features* that are shared by most OO languages and that, together, support a *distinctive and useful* programming style.

Dynamic dispatch

Perhaps the most basic characteristic of object-oriented programming is *dynamic dispatch*: when an operation is invoked on an object, the ensuing behavior depends on the object itself, rather than being fixed once and for all (as when we apply a function to an argument).

Two objects of the *same type* (i.e., responding to the same set of operations) may be implemented internally in *completely different* ways.

Example (in Java)

```
class A {
        int x = 0;
        int m() { x = x+1; return x; }
        int n() { x = x-1; return x; }
}
class B extends A {
        int m() { x = x+5; return x; }
}
class C extends A {
        int m() { x = x-10; return x; }
}
```

Note that `(new B()).m()` and `(new C()).m()` invoke completely different code!

Encapsulation

In most OO languages, each object consists of some internal state *encapsulated* with *a collection of method implementations* operating on that state.

- state directly accessible to methods
- state invisible / inaccessible from outside the object

Aside: encapsulation

Encapsulation is arguably a little less fundamental than dynamic dispatch, in the sense that there are several OO languages (e.g., CLOS, Dylan, and Cecil) that do *not* encapsulate state with methods.

These languages are based, instead, on *multi-methods*, a form of *ad-hoc polymorphism*.

Although their basic mechanisms are quite different, the higher-level programming idioms (classes, inheritance, etc.) arising in multi-method languages are surprisingly similar to those in “mainstream” OO languages.

Encapsulation

In Smalltalk, encapsulation is mandatory; whereas in Java, encapsulation of internal state is optional. For full encapsulation, fields must be marked protected:

```
class A {
        protected int x = 0;
        int m()  { x = x+1; return x; }
        int n()   { x = x-1;  return x; }
}
class B extends A {
        int m()  { x = x+5;  return x; }
}
class C extends A {
        int m()  { x = x-10;  return x; }
}
```

The code `(new B()). x` is not allowed.

Aside: Objects vs. ADTs

An ADT comprises:

- A *hidden* representation type X
- A collection of operations for creating and manipulating elements of type X

Similar to OO encapsulation in that only the operations provided by the ADT are allowed to directly manipulate elements of the abstract type. But *different* in that there is just one (hidden) representation type and *just one implementation of the operations* — no dynamic dispatch.

Both styles have advantages.

N.B. : in the OO community, the term "*abstract data type*" is often used as more or less a synonym for "object type." This is unfortunate, since it confuses two *rather different concepts*.

Subtyping and Encapsulation

The “type” (or “interface” in Smalltalk terminology) of an object is just *the set of operations* that can be performed on it (and the types of their parameters and results); it does not include the internal representation.

Object interfaces fit naturally into a *subtype relation*.

- *An interface listing more operations is “better” than one listing fewer operations.*

This gives rise to a natural and useful form of *polymorphism*: we can write one piece of code that operates uniformly on any object whose interface is “at least as good as I” (i.e., any object that supports at least the operations in I).

Example

```
// ... class A and subclasses B and C as above...

class D {
    int p (A myA)  { return myA.m(); }
}
...
D d = new D();
int z = d.p (new B());
int w = d.p (new C());
```

Inheritance

Objects that share parts of their interfaces will typically (though not always) share parts of their behaviors.

To avoid duplication of code, the way is to write the implementations of these behaviors in *just one place*.

⟹ *inheritance*

Inheritance

Basic mechanism of inheritance: *classes*

A class *is a data structure* that can be

- *instantiated* to create new objects ("instances")
- *refined* to create new classes ("subclasses")

N.B.: some OO languages offer *an alternative mechanism*, called *delegation*, which allows new objects to be derived by refining the behavior of existing objects.

Example

```
class A {
        protected int x = 0;
        int m() { x = x+1; return x; }
        int n() { x = x-1; return x; }
}
class B extends A {
        int  p() { x = x*10; return x; }
}
```

An instance of B has methods m, n, and p. The first two are inherited from A.

Late binding/open recursion

Most OO languages offer an extension of the basic mechanism of classes and inheritance called *late binding* or *open recursion*.

Late binding allows a method within a class to call another method via a *special "pseudo-variable"* this. If the second method is overridden by some subclass, then the behavior of the first method automatically changes as well.

Though quite useful in many situations, late binding is rather tricky, both to define (as we will see) and to use appropriately. For this reason, it is sometimes *deprecated in practice*.

Examples

```
class E {
        protected int x = 0;
        int m()  { x = x+1; return x; }
        int n()  { x = x-1;  return this.m(); }
}

class F extends E {
        int m()  { x = x+100;  return x; }
}
```

Q:

- What does `(new E()).n()` return?
- What does `(new F()).n()` return?

Calling “super”

It is sometimes convenient to “re-use” the functionality of an overridden method.

Java provides a mechanism called super for this purpose.

Example

```
class E {
        protected int x = 0;
        int m() { x = x+1; return x; }
        int n() { x = x-1; return this.m(); }
}

class G extends E {
        int m() { x = x+100; return super.m(); }
}
```

What does (new G()).n() return?

Getting down to details (in the lambda-calculus)...

Objects

A *data structure*

- encapsulating some internal state
- offering access to this state

via a *collection of methods*.

The *internal state* is typically organized as a number of mutable instance variables that are shared among the methods and inaccessible to the outsiders.

Simple objects with encapsulated state

```
class Counter {
        protected int x = 1;                        // Hidden state
        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();
```

How do we encode objects in the lambda-calculus?

Objects built with λ-calculus

$$\begin{aligned}
c \;=\; & \text{let } x = \text{ref } 1 \text{ in} \\
& \quad \{\, \text{get} \;=\; \lambda\_{:}\,\text{Unit}. \quad !\,x, \\
& \quad \;\; \text{inc} \;=\; \lambda\_{:}\,\text{Unit}. \quad x := \text{succ}(!\,x)\}; \\
\Longrightarrow \; & c : \text{Counter}
\end{aligned}$$

where

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

The abstraction of block evaluation of the method bodies *when the object is created*.

- Allowing the bodies to be *evaluated repeatedly*

Using Objects

$$\mathrm{inc3} = \lambda c{:}\,\mathrm{Counter}.\,(c.\mathrm{inc}\ \mathrm{unit};\ c.\mathrm{inc}\ \mathrm{unit};\ c.\mathrm{inc}\ \mathrm{unit});$$

$$\Longrightarrow\ \mathrm{inc3} : \mathrm{Counter} \longrightarrow \mathrm{Unit}$$

$$(\mathrm{inc3}\ c;\ \mathrm{inc3}\ c;\ c.\mathrm{get}\ \mathrm{unit});$$

$$\Longrightarrow\ 7{:}\ \mathrm{Nat}$$

Object Generators

$$\begin{aligned}
&\text{newCounter} = \\
&\qquad \lambda\_:\text{Unit}.\ \text{let}\ x = \text{ref}\ 1\ \text{in} \\
&\qquad\qquad \{\text{get} = \lambda\_:\text{Unit}.\ !x, \\
&\qquad\qquad\ \ \text{inc} = \lambda\_:\text{Unit}.\ x := \text{succ}(!x)\}; \\
&\Longrightarrow \text{newCounter} : \text{Unit} \longrightarrow \text{Counter}
\end{aligned}$$

a function that creates and returns a new counter *every time it is called*.

Grouping Instance Variables

Rather than a single reference cell, the states of most objects consist of a number of *instance variables* or *fields*.

It will be convenient (later) to group these into a single record (as a single unit).

$$
\begin{aligned}
&\text{newCounter} \;=\\
&\qquad \lambda\_{:}\,\text{Unit}.\;\; \text{let}\;\, r \;=\; \boxed{\{x = \text{ref}\;1\}}\; \text{in}\\
&\qquad\qquad \{\, \text{get} \;=\; \lambda\_{:}\,\text{Unit}.\; !\,(r.x),\\
&\qquad\qquad\;\; \text{inc} \;=\; \lambda\_{:}\,\text{Unit}.\;\; r.x := \text{succ}(!\,(r.x))\};
\end{aligned}
$$

The local variable r has type of *representation type*

$$\text{CounterRep} \;=\; \{x{:}\;\; \text{Ref Nat}\}$$

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 <: Counter

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

Subtyping

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

$$\begin{aligned}
&\text{newResetCounter} = \\
&\quad \lambda\_: \text{Unit}.\ \text{let } r = \{x = \text{ref } 1\} \text{ in} \\
&\qquad \{\, \text{get} = \lambda\_: \text{Unit}.\ !\,(r.x), \\
&\qquad\ \ \text{inc} = \lambda\_: \text{Unit}.\ r.x := \text{succ}(!\,(r.x)), \\
&\qquad\ \ \text{reset} = \lambda\_: \text{Unit}.\ r.x := 1\};
\end{aligned}$$

$$\Longrightarrow \ \text{newResetCounter} : \text{Unit} \longrightarrow \text{ResetCounter}$$

Subtyping

rc = newResetCounter unit;

(inc3 rc; rc. reset unit; inc3 rc; rc. get unit);

⟹ 4: Nat

Simple Classes

The definitions of newCounter and newResetCounter are identical except for the reset method.

This violates a basic principle of software engineering:

> *Each piece of behavior should be implemented in just one place in the code.*

Reusing Methods

Idea: could we just re-use the methods of some existing object to build a new object?

$$
\begin{array}{l}
\mathrm{resetCounterFromCounter} \;=\\
\quad \lambda c{:}\,\mathrm{Counter}.\ \mathrm{let}\ r \;=\; \{x \;=\; \mathrm{ref}\ 1\}\ \mathrm{in}\\
\qquad\quad \{\,\mathrm{get} \;=\; c.\mathrm{get},\\
\qquad\qquad \mathrm{inc} \;=\; c.\mathrm{inc},\\
\qquad\qquad \mathrm{reset} \;= \lambda\_{:}\,\mathrm{Unit}.\ r.x := 1\};
\end{array}
$$

Reusing Methods

Idea: could we just re-use the methods of some existing object to build a new object?

$$
\begin{array}{l}
\mathrm{resetCounterFromCounter} \;=\\
\quad \lambda c{:}\,\mathrm{Counter}.\;\; \mathrm{let}\; r \;=\; \{x \;=\; \mathrm{ref}\; 1\}\; \mathrm{in}\\
\qquad\quad \{\,\mathrm{get} \;=\; c.\mathrm{get},\\
\qquad\qquad \mathrm{inc} \;=\; c.\mathrm{inc},\\
\qquad\qquad \mathrm{reset} \;=\lambda\_{:}\,\mathrm{Unit}.\;\; r.x := 1\};
\end{array}
$$

No: This doesn’t work properly because the reset method does not have access to the local variable r of the original counter.

$\Longrightarrow \quad \mathrm{classes}$

Classes

A class is a run-time data structure that can be

1. *instantiated* to yield new objects
2. *extended* to yield new classes

Classes

To avoid the problem we observed before, what we need to do is to *separate the definition of the methods*

$$
\begin{aligned}
&\text{counterClass} \;=\\
&\quad \lambda r\colon \text{CounterRep}.\\
&\qquad \{\,\text{get} \;=\; \_\colon \text{Unit}.\,!\,(r.x),\\
&\qquad\;\; \text{inc} \;=\; \_\colon \text{Unit}.\, r.x := \text{succ}(!\,(r.x))\};\\
&\Longrightarrow\; \text{counterClass} : \;\text{CounterRep} \longrightarrow \text{Counter}
\end{aligned}
$$

from *the act of binding these methods to a particular set of instance variables*:

$$
\begin{aligned}
&\text{newCounter} \;=\\
&\quad \lambda\_\colon \text{Unit}.\,\text{let } r \;=\; \{x = \text{ref } 1\} \text{ in}\\
&\qquad\quad \text{counterClass } r;\\
&\Longrightarrow\; \text{newCounter}\colon \text{Unit} \longrightarrow \text{Counter}
\end{aligned}
$$

Defining a Subclass

$$
\begin{array}{l}
\mathrm{resetCounterClass} \;=\\
\quad \lambda r{:}\,\mathrm{CounterRep}.\\
\qquad \mathrm{let\ super} \;=\; \mathrm{counterClass}\;\; r \;\mathrm{in}\\
\qquad \{\; \mathrm{get} \;=\; \mathrm{super.get},\\
\qquad\quad \mathrm{inc} \;=\; \mathrm{super.inc},\\
\qquad\quad \mathrm{reset} \;=\lambda\_{:}\,\mathrm{Unit}.\;\; r.x := 1\};
\end{array}
$$

$\Longrightarrow$ $\mathrm{resetCounterClass} : \mathrm{CounterRep} \longrightarrow \mathrm{ResetCounter}$

$$
\begin{array}{l}
\mathrm{newResetCounter} \;=\\
\qquad \lambda\_{:}\,\mathrm{Unit}.\;\; \mathrm{let}\; r \;=\; \{x = \mathrm{ref}\; 1\}\; \mathrm{in\ resetCounterClass}\;\; r;
\end{array}
$$

$\Longrightarrow$ $\mathrm{newResetCounter} : \mathrm{Unit} \longrightarrow \mathrm{ResetCounter}$

Overriding and adding 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

In general, when we define a subclass we will want *to add new instances variables* to its representation.

$$\begin{aligned}
&\mathrm{BackupCounter} \;=\; \{\,\mathrm{get}\colon \mathrm{Unit} \longrightarrow \mathrm{Nat}, \qquad \mathrm{inc}\colon \mathrm{Unit} \longrightarrow \mathrm{Unit},\\
&\qquad\qquad\qquad\qquad\qquad \mathrm{reset}\colon \mathrm{Unit} \longrightarrow \mathrm{Unit}, \;\; \mathrm{backup}\colon\; \mathrm{Unit} \longrightarrow \mathrm{Unit}\};\\
&\mathrm{BackupCounterRep} \;=\; \{\mathrm{x}\colon\; \mathrm{Ref\ Nat}, \boxed{\mathrm{b}\colon\; \mathrm{Ref\ Nat}}\};
\end{aligned}$$

$$\begin{aligned}
&\mathrm{backupCounterClass} \;=\\
&\quad \lambda \mathrm{r}\colon \mathrm{BackupCounterRep}.\\
&\qquad \mathrm{let\ super} \;=\; \mathrm{resetCounterClass\ r\ in}\\
&\qquad\quad \{\,\mathrm{get} \;=\; \mathrm{super.get},\\
&\qquad\quad\;\; \mathrm{inc} \;=\; \mathrm{super.inc},\\
&\qquad\quad\;\; \mathrm{reset} \;=\lambda\_\colon \mathrm{Unit}.\quad \boxed{\begin{array}{l}\mathrm{r.x} := !(\mathrm{r.b}),\\ \;\mathrm{r.b} := !(\mathrm{r.x})\};\end{array}}\\
&\qquad\quad\;\; \mathrm{backup} \;=\lambda\_\colon \mathrm{Unit}.
\end{aligned}$$

$\Longrightarrow$ backupCounterClass : BackupCounterRep ⟶ BackupCounter

Aside

Notes:

- backupCounterClass both *extends* (with backup) and *overrides* (with a new reset) the definition of counterClass
- subtyping is essential here (in the definition of super)

$$
\begin{array}{l}
\mathrm{backupCounterClass} \;=\\
\quad \lambda r{:}\,\mathrm{BackupCounterRep}.\\
\qquad \mathrm{let}\ \mathrm{super} \;=\; \mathrm{resetCounterClass}\ r\ \ \mathrm{in}\\
\qquad\quad \{\mathrm{get} \;=\; \mathrm{super.get},\\
\qquad\quad\ \ \mathrm{inc} \;=\; \mathrm{super.inc},\\
\qquad\quad\ \ \mathrm{reset} \;=\lambda\_{:}\,\mathrm{Unit}.\, r.x := !(r.b),\\
\qquad\quad\ \ \mathrm{backup} \;=\lambda\_{:}\,\mathrm{Unit}.\, r.b := !(r.x)\};
\end{array}
$$

Calling super

Suppose (for the sake of the example) that we wanted every call to inc to first *back up the current state*. We can avoid copying the code for backup by making inc use the backup and inc methods from super.

$$
\begin{aligned}
&\mathrm{funnyBackupCounterClass} \ = \\
&\qquad\qquad \lambda r{:}\ \mathrm{BackupCounterRep}. \\
&\qquad\qquad\quad \mathrm{let\ super} \ = \ \mathrm{backupCounterClass}\ r\ \mathrm{in} \\
&\qquad\qquad\qquad \{\mathrm{get} \ = \ \mathrm{super.get}, \\
&\qquad\qquad\qquad\ \ \mathrm{inc} \ = \lambda\_{:}\ \mathrm{Unit}.\ \boxed{(\mathrm{super.backup\ unit};\ \ \mathrm{super.inc\ unit})}, \\
&\qquad\qquad\qquad\ \ \mathrm{reset} \ = \ \mathrm{super.reset}, \\
&\qquad\qquad\qquad\ \ \mathrm{backup} \ = \ \mathrm{super.backup}\};
\end{aligned}
$$

$\Longrightarrow$

$$\mathrm{funnyBackupCounterClass} : \ \mathrm{BackupCounterRep} \longrightarrow \mathrm{BackupCounter}$$

Calling between methods

What if counters have set, get, and inc methods:

$$\begin{aligned}\mathrm{SetCounter} \;=\; \{&\mathrm{get}\colon \mathrm{Unit} \longrightarrow \mathrm{Nat}, \mathrm{set}\colon \mathrm{Nat} \longrightarrow \mathrm{Unit},\\ &\mathrm{inc}\colon \mathrm{Unit} \longrightarrow \mathrm{Unit}\};\end{aligned}$$

$$\begin{aligned}&\mathrm{setCounterClass} \;=\\ &\quad \lambda r\colon \mathrm{CounterRep}.\\ &\qquad \{\, \mathrm{get} \;=\lambda\_\colon \mathrm{Unit}.\;\; !(r.x),\\ &\qquad\;\; \mathrm{set} \;=\lambda i\colon \mathrm{Nat}.\;\; r.x := i,\\ &\qquad\;\; \mathrm{inc} \;=\lambda\_\colon \mathrm{Unit}.\;\; r.x := (\mathrm{succ}\; r.x)\,\};\end{aligned}$$

Calling between methods

What if counters have set, get, and inc methods:

$$\begin{aligned}\mathrm{SetCounter} \;=\; \{&\mathrm{get}: \mathrm{Unit} \longrightarrow \mathrm{Nat}, \qquad \mathrm{set}: \mathrm{Nat} \longrightarrow \mathrm{Unit},\\ &\mathrm{inc}: \mathrm{Unit} \longrightarrow \mathrm{Unit}\};\end{aligned}$$

$$\begin{aligned}&\mathrm{setCounterClass} \;=\\ &\quad \lambda \mathrm{r}: \mathrm{CounterRep}.\\ &\qquad \{\,\mathrm{get} \;=\lambda\_: \mathrm{Unit}.\;\; !\,(\mathrm{r.x}),\\ &\qquad\;\; \mathrm{set} \;=\lambda \mathrm{i}: \mathrm{Nat}.\;\; \mathrm{r.x} := \mathrm{i},\\ &\qquad\;\; \boxed{\mathrm{inc} \;=\lambda\_: \mathrm{Unit}.\;\; \mathrm{r.x} := (\mathrm{succ\ r.x})\,\};}\end{aligned}$$

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?

Calling between methods

In Java we would write:

```
class SetCounter {
        protected int x  =  0;
        int get () { return x;  }
        void set (int i) { x  =  i; }
        void inc () { this. set( this. get()  +  1 );  }
}
```

Better ?

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

Check: the type of the inner λ-abstraction is SetCounter⟶ SetCounter, so the type of the fix expression is SetCounter.

This is just a definition of a group of *mutually recursive functions*.

Better…

Note that the *fixed point* in

$$\begin{array}{l}
\mathrm{setCounterClass}\ = \\
\quad \lambda r\colon \mathrm{CounterRep}. \\
\qquad \mathrm{fix} \\
\qquad\ (\lambda \mathrm{this}\colon\ \mathrm{SetCounter}. \\
\qquad\qquad \{\mathrm{get}\ = \lambda\_\colon \mathrm{Unit}.\ !\,(r.x), \\
\qquad\qquad\ \ \mathrm{set}\ = \lambda i\colon \mathrm{Nat}.\ r.x := i, \\
\qquad\qquad\ \ \mathrm{inc}\ = \lambda\_\colon \mathrm{Unit}.\ \ \mathrm{this.set}\ \ (\mathrm{succ}\ (\mathrm{this.get\ unit}))\});
\end{array}$$

is “closed” — we “tie the knot” when we build the record (arranging that the very record we are constructing is the one passed as this) , and the use of fix is entirely internal to setCounterClass

So this does not model the behavior of this (or self) in real OO languages (Most OO languages actually support a more general form of recursive call between methods, as open recursion or late binding of self).

Better…

Idea: move the application of fix from *the class definition*…

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

... to *the object creation function*:

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

In essence, we are switching the order of fix and λr: CounterRep…

Better...

Note that we have changed the types of classes from...

```
setCounterClass  =
    λr: CounterRep.
        fix
         (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
```

... to :

```
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
```

Using this

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.

$$\begin{aligned}\mathrm{InstrCounter} &= \{\mathrm{get}\colon \mathrm{Unit} \longrightarrow \mathrm{Nat},\quad \mathrm{set}\colon \mathrm{Nat} \longrightarrow \mathrm{Unit},\\ &\qquad\qquad \mathrm{inc}\colon \mathrm{Unit} \longrightarrow \mathrm{Unit}, \mathrm{accesses}\colon \mathrm{Unit} \longrightarrow \mathrm{Nat}\};\\ \mathrm{InstrCounterRep} &= \{\mathrm{x}\colon\ \mathrm{Ref\ Nat}, \mathrm{a}\colon\ \mathrm{Ref\ Nat}\};\end{aligned}$$

Using this

```
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
```

Notes:

- the methods use both this (which is passed as a parameter) and super (which is constructed using this 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

More refinement …

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

will cause the evaluator to *diverge*!

Intuitively, the problem is the "*unprotected*" use of this in the call to setCounterClass in instrCounterClass:

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

A small fly in the ointment

To see why this diverges, consider a simpler example:

$$\begin{aligned} \mathrm{ff} &= \lambda \mathrm{f}{:}\,\mathrm{Nat} \longrightarrow \mathrm{Nat}.\\ &\quad\ \mathrm{let}\ \mathrm{f}' = \mathrm{f}\ \ \mathrm{in}\\ &\quad\ \lambda \mathrm{n}{:}\,\mathrm{Nat}.\ 0 \end{aligned}$$

$$\Longrightarrow\quad \mathrm{ff} : (\mathrm{Nat} \longrightarrow \mathrm{Nat}) \longrightarrow (\mathrm{Nat} \longrightarrow \mathrm{Nat})$$

Now:

$$\begin{aligned} \mathrm{fix}\ \mathrm{ff} &\longrightarrow \mathrm{let}\ \mathrm{f}' = (\mathrm{fix}\ \mathrm{ff})\ \ \mathrm{in}\ \lambda \mathrm{n}{:}\,\mathrm{Nat}.\ 0\\ &\longrightarrow \mathrm{let}\ \mathrm{f}' = \mathrm{ff}\ (\mathrm{fix}\ \mathrm{ff})\ \ \mathrm{in}\ \lambda \mathrm{n}{:}\,\mathrm{Nat}.\ 0\\ &\longrightarrow \text{uh oh} \ldots \end{aligned}$$

Intuitively, the problem here is that the argument to the fix operator is using its own argument, self, too early. The operational semantics of fix is defined with the expectation that, when we apply fix to some function $\lambda x.\,t$, the body t should refer to x only in protected positions.

One possible solution

Idea: “delay” this by putting a *dummy abstraction* in front of it…

$$
\begin{aligned}
&\text{setCounterClass} \;= \\
&\quad \lambda r{:}\,\text{CounterRep}. \\
&\quad \lambda \text{this}{:}\;\text{Unit} \longrightarrow \text{SetCounter}. \\
&\qquad\quad \lambda\_{:}\,\text{Unit}. \\
&\qquad\qquad\qquad \{\text{get} \;=\; \lambda\_{:}\,\text{Unit}.\;\; !\,(r.x), \\
&\qquad\qquad\qquad\;\; \text{set} \;=\; \lambda i{:}\,\text{Nat}.\;\;\; r.x := i, \\
&\qquad\qquad\qquad\;\; \text{inc} \;=\; \lambda\_{:}\,\text{Unit}.\;\; (\text{this unit}).\text{set} \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\text{succ}((\text{this unit}).\text{get unit}))\}; \\
&\Longrightarrow\;\; \text{setCounterClass} : \\
&\;\text{CounterRep} \longrightarrow (\text{Unit} \longrightarrow \text{SetCounter}) \longrightarrow (\text{Unit} \longrightarrow \text{SetCounter})
\end{aligned}
$$

$$
\begin{aligned}
&\text{newSetCounter} \;= \\
&\quad \lambda\_{:}\,\text{Unit}.\;\; \text{let } r \;=\; \{x = \text{ref } 1\} \text{ in} \\
&\qquad\qquad\qquad \text{fix } (\text{setCounterClass } r) \text{ unit};
\end{aligned}
$$

One possible solution

Similarly:

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

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

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

Recap

Multiple representations

All the objects we have built in this series of examples have type Counter.

However, 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. (lexical scoping ensures that 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.

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 this (which should have the same interface and be based on the same record of instance variables)
- uses r and this to instantiate its superclass
- constructs a record of method implementations, copying some directly from super and implementing others in terms of this and super.

The this parameter is "resolved" *at object creation time* using fix.