CIS 500
Software Foundations
Fall 2006

November 22

Continuing with Objects

Calling between methods

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

```
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) });
```

Calling between methods

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

```
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) });
```

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.

Note that the fixed point in

```
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))});
```

is "closed" — we "tie the knot" when we build the record.

So this does *not* model the behavior of `this` (or `self`) in real OO languages.

Idea: move the application of `fix` from the class definition...

```
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))};
```

...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...`

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))});
```

$\Longrightarrow$ `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))};
```

$\Longrightarrow$

`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.

```
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)};
```

$\Longrightarrow$ `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`

One 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 (see TAPL for details), 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
         ...
```

To see why this diverges, consider a simpler example:

```
  ff = λf:Nat→Nat.
         let f′ = f in
         λn:Nat. 0
⟹ ff : (Nat→Nat) → (Nat→Nat)
```

Now:

```
     fix ff   ⟶  let f′ = (fix ff) in λn:Nat. 0
              ⟶  let f′ = ff (fix ff) in λn:Nat. 0
              ⟶  uh oh...
```

One possible solution

Idea: "delay" `this` by putting a dummy abstraction in front of it...

```
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((this unit).get unit))};
⟹ setCounterClass :
      CounterRep → (Unit→SetCounter) → (Unit→SetCounter)

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

Similarly:

```
instrCounterClass =
  λr:InstrCounterRep.
  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)};

newInstrCounter =
  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 TAPL 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`.

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`.

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`.

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.]