Self-Reference

Lecture 16
CS 565
4/7/06
Method invocations

- Consider a class defining counters with get, set, and inc methods:

  \[\text{setCounterClass} = \lambda r: \text{CounterRep.}\]
  \[
  \{ \text{get} = \lambda \_: \text{Unit. !}(r \cdot x), \\
  \text{set} = \lambda \_: \text{Unit. } r.x := 1, \\
  \text{inc} = \lambda \_: \text{Unit. } r.x := (\text{succ } r.x) \} \]

- Bad style: can express inc in terms of get and set

- Would like to avoid repeating implementation of this functionality.
Method invocations

setCounterClass =
    \ r: CounterRep.
    fix
    (\ self: SetCounter.
        { get = \ _ : Unit. !(r x),
          set = \ _ : Unit. r.x := 1,
          inc = \ _ : Unit.
              self.set (succ (self.get unit)) }
    )

- The type of the inner \ abstraction is SetCounter \ SetCounter so the type of the object returned by the fix expression is SetCounter
- The type of setCounterClass is CounterRep \ SetCounter
  - SetCounter is a record type that corresponds to the record returned by the inner abstraction.
- Define a set of mutually recursive functions.
Understanding Self

- Note that the fixed point in setCounterClass:

  \[ \lambda \text{r}: \text{CounterRep.} \]
  
  \[ \text{fix} \]
  
  \[ (\lambda \text{self}: \text{SetCounter.} \]
  
  \[ \{ \text{get} = \lambda \_ : \text{Unit.} \;!\!(\text{r} \times), \]
  
  \[ \text{set} = \lambda \_ : \text{Unit.} \;\text{r}\.x := 1, \]
  
  \[ \text{inc} = \lambda \_ : \text{Unit.} \]
  
  \[ \text{self.set} \]
  
  \[ (\text{succ} \;\text{self.get unit}) \} \]
  
  is closed - the recursion is closed when we build the record

- This does not model the behavior of self or this found in real object oriented languages. Why?
**Another Approach**

- Idea: Move the application of fix from the class definition.

```
setCounterClass = λ r: CounterRep.
                (λ self: SetCounter.
                 { get = λ _ : Unit. !(r x),
                   set = λ i: Nat. r.x := i,
                   inc = λ _ : Unit.
                     self.set (succ (self.get unit)) }
                
             to the object creation function:
             newSetCounter = λ _ : Unit. let r = {x=ref 1} in
                fix (setCounterClass r)
```

**In essence:** switch the order of fix and λ r: CounterRep
Types

The types have changed from:

```
setCounterClass =

λ r: CounterRep.

fix

(λ self: SetCounter.

{ get = λ _ : Unit. !(r x),
  set = λ _ : Unit. r.x := 1,
  inc = λ _ : Unit.
      self.set (succ (self.get unit)) })

→ setCounterClass: CounterRep → SetCounter
```
Types

setCounterClass =

\( \lambda \) r: CounterRep.

(\( \lambda \) self: SetCounter.

\{ get = \( \lambda \) : Unit. !(r x),

set = \( \lambda \) i: Nat. r.x := i,

inc = \( \lambda \) : Unit.

self.set (succ (self.get unit)) \}

\( \Rightarrow \) setCounterClass: CounterRep \( \rightarrow \) SetCounter \( \rightarrow \) SetCounter)
Using Self

Consider a new class of counter objects defined to be a subclass of set-counters that keeps a record of the number of times a counter is set:

\[
\text{InstrCounter} = \{ \text{get} : \text{Unit} \rightarrow \text{Nat}, \\
\quad \text{set} : \text{Nat} \rightarrow \text{Unit}, \\
\quad \text{inc} : \text{Unit} \rightarrow \text{Unit}, \\
\quad \text{accesses} : \text{Unit} \rightarrow \text{Nat}\}
\]

\[
\text{InstrCounterRep} = \{ x : \text{Nat ref}, a : \text{Nat ref}\}
\]
Implementation

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

- The methods in instrCounterClass use both self (passed as a parameter) and super (constructed using self and the representation).
- The definition of inc in super will invoke the set and get methods defined here which in turn calls set.
- Subtyping plays a crucial role here in the call to setCounterClass (how?)
Issues

Consider how an instance of an instrumented counter is created:

\[\lambda \_ : \text{Unit. let } r = \{ x = \text{ref 1, a = ref 0} \} \text{ in} \]
\[\text{fix (instrCounterClass r)}\]

Problem: the construction of super happens in an “unprotected” piece of code (not encapsulated by an abstraction):

\[\text{instrCounterClass =} \]
\[\lambda r: \text{InstrCounterRep.} \]
\[\lambda \text{self: InstrCounter.} \]
\[\text{let super = setCounterClass } r \text{ self in ...} \]

What happens here?
Example

\[ ff = \lambda f: \text{Nat} \to \text{Nat} \]

\[ \text{let } f' = f \]
\[ \text{in } \lambda n: \text{Nat}. 0 \]

\[ \Rightarrow ff : (\text{Nat} \to \text{Nat}) \to (\text{Nat} \to \text{Nat}) \]

But, \( \text{fix } ff \Rightarrow ff (\text{fix } ff) \)
\[ \Rightarrow \text{let } f' = (\text{fix } ff) \text{ in } ... \]
\[ \Rightarrow \text{let } f' = ff (\text{fix } ff) \text{ in } ... \]
\[ \Rightarrow ? \]
Eager Evaluation of Super

- When we apply \( \text{fix (instrCounterClass } r) \)
  - We evaluate
    \[
    \text{fix (} \lambda \text{self:InstrCounterClass}
    \]
    \[
    \text{let super = setCounterClass } r \text{ self in}
    \]
    ...  
  - By evaluation rule for fix, this yields
    \[
    \text{let super = setCounterClass } r \text{ (fix } \lambda \text{self... in)}
    \]
    However, to reduce the application of setCounterClass requires us to reduce (fix \( \lambda \text{self...} \)) to a value. The current structure of the InstrCounterClass will not permit that.
    Intuitively, self is being applied to fix too early.
Delay evaluation of self via a dummy abstraction:

\[
\text{setCounterClass} = \\
\lambda \ r: \text{CounterRep.} \\
\quad (\lambda \ self: \text{Unit} \to \text{SetCounter.} \\
\quad \lambda \ _: \text{Unit.} \\
\quad \{ \text{get} = \lambda : \text{Unit.} ! (r \ x), \\
\quad \text{set} = \lambda \ i: \text{Nat.} \ r.x := i, \\
\quad \text{inc} = \lambda \ _: \text{Unit.} \\
\quad (\text{self unit}).\text{set} (\text{succ} ((\text{self unit}).\text{get} \ \text{unit})))\}
\]

\[
\text{setCounterClass}: \text{CounterRep} \to (\text{Unit} \to \text{SetCounter}) \to \text{SetCounter}
\]
Remedy

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

\[
\text{newInstrCounter} = \\
\lambda _ : \text{Unit}. \text{let} r = \{x = \text{ref 1}, a = \text{ref 0}\}
\text{in} \text{fix (instrCounterClass r) unit}
\]
Evaluation

- This approach is correct in that we can instantiate `instrCounterClass` (without diverging).
- However, delaying the evaluation of `self` has the unfortunate effect of "recomputing" the object definition every time `self` is evaluated.
  - Are there better approaches?
Implementing Self

- Main problem with previous approach is that methods to self are recomputed every time a call is made.
- Two alternatives:
  - Use a different implementation strategy, e.g., use references instead of fixpoints.
  - Abandon the notion of encoding objects directly in the $\lambda$-calculus, developing instead an alternative calculus in which objects and classes are primitives.
Using References

- Intuition: instead of abstracting a record of methods that is created using fix, abstract a reference to a record of methods and allocate this record first.

```latex
setCounterClass =
  \lambda r: \text{CounterRep}.
    \lambda self: \text{SetCounter ref}.
      \{get = \lambda _: \text{Unit}. !(r.x),
        set = \lambda i: \text{Nat}. r.x := i,
        inc = \lambda _: \text{Unit}. (!self).set (\text{succ} ((!self).get \text{unit}))\}
```

The self parameter is a reference to a cell that contains the method of the current object.
To create a counter, we first create a dummy counter and then subsequently set it:

dummySetCounter =
{ get = \_ : Unit. 0,
  set = \_ i:Nat. unit,
  inc = \_ : Unit. unit}

newSetCounter =
\_ : Unit.
  let r = {x=ref 1}
   c = ref DummySetCounter
  in (c := (setCounterClass r c); !c)

Since all dereferences to self are protected inside an abstraction, the contents of the dummy counter will never be accessed.

What is the general problem with using references to model inheritance and subclassing?