

Self-Reference

Lecture 15

CS 565

4/01/09

Method invocations

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

`setCounterClass =`

`λ r: CounterRep.`

`{ get = λ _: Unit. !(r x),`

`set = λ n: Unit. r.x := n,`

`inc = λ _: Unit. r.x := (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 =  
  λ r: CounterRep.  
    fix  
      (λ self: SetCounter.  
        { get = λ _ : Unit. !(r x),  
          set = λ n : Unit. r.x := n,  
          inc = λ _ : Unit.  
              self.set (succ (self.get unit)) })
```

- The type of the inner λ abstraction is $\text{SetCounter} \rightarrow \text{SetCounter}$ so the type of the object returned by the `fix` expression is `SetCounter`
 - The type of `setCounterClass` is $\text{CounterRep} \rightarrow \text{SetCounter}$
 - `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`:

`λ r: CounterRep.`

`fix`

`(λ self: SetCounter.`

`{ get = λ _ : Unit. !(r x),`

`set = λ n : Unit. r.x := n,`

`inc = λ _ : Unit.`

`self.set`

`(succ (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 n: Nat. r.x := n,`

`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 n : Unit. r.x := n,
```

```
        inc = λ _ : Unit.
```

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

→ setCounterClass: CounterRep → SetCounter

Types

setCounterClass =

λ r: CounterRep.

(λ self: SetCounter.

{ get = λ : Unit. !(r x),

set = λ n: Nat. r.x := n,

inc = λ : Unit.

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

→ setCounterClass: CounterRep → SetCounter → 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:

$$\text{InstrCounter} = \{ \text{get} : \text{Unit} \rightarrow \text{Nat}, \\ \text{set} : \text{Nat} \rightarrow \text{Unit}, \\ \text{inc} : \text{Unit} \rightarrow \text{Unit}, \\ \text{accesses} : \text{Unit} \rightarrow \text{Nat} \}$$
$$\text{InstrCounterRep} = \{ x : \text{Nat ref}, a : \text{Nat ref} \}$$

Implementation

instrCounterClass =

λ r: InstrCounterRep.

λ self: InstrCounter.

let super = setCounterClass r self

in { get = super.get,

set = λ n: Nat.

(r.a := succ(! r a); super.set n),

inc = super.inc,

accesses = λ _ : Unit. !(r a) }

→ instrCounterClass : InstrCounterRep → InstrCounter →
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:

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

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

```
instrCounterClass =  
  λ r: InstrCounterRep.  
    λ self: InstrCounter.  
      let super = setCounterClass r self  
      in ...
```

What happens here?

Example

$ff = \lambda f: \text{Nat} \rightarrow \text{Nat}.$

$\text{let } f' = f$

$\text{in } \lambda n: \text{Nat}. 0$

→ $ff : (\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \rightarrow \text{Nat})$

But, $\text{fix } ff \Rightarrow ff (\text{fix } ff)$

$\Rightarrow \text{let } f' = (\text{fix } ff) \text{ in } \dots$

$\Rightarrow \text{let } f' = ff (\text{fix } ff) \text{ in } \dots$

Eager Evaluation of Super

- When we apply `fix (instrCounterClass r)`

- We evaluate

```
fix (λself:InstrCounterClass
```

```
    let super = setCounterClass r self in
```

```
    ...
```

- By evaluation rule for `fix`, this yields

```
let super = setCounterClass r (fix λself...) in
```

However, to reduce the application of `setCounterClass` requires us to reduce `(fix λself...)` to a value. The current structure of the `InstrCounterClass` will not permit that.

Intuitively, `self` is being applied to `fix` too early.

Remedy

Delay evaluation of self via a dummy abstraction:

```
setCounterClass =  
  λ r: CounterRep.  
    (λ self: Unit → SetCounter.  
      λ _ : Unit.  
        { get = λ : Unit. !(r x),  
          set = λ n: Nat. r.x := n,  
          inc = λ _ : Unit.  
            (self unit).set (succ ((self unit).get unit))})  
setCounterClass: CounterRep → (Unit → SetCounter) →  
  SetCounter
```

Remedy

```
instrCounterClass =  
  λ r: InstrCounterRep.  
    λ self: Unit → InstrCounter.  
      λ _ : Unit.  
        let super = setCounterClass r self unit  
        in { get = super.get,  
            set = λ n: Nat.  
                  (r.a := succ(! r a); super.set n),  
            inc = super.inc,  
            accesses = λ : Unit. !(r a)
```

```
newInstrCounter =  
  λ _ : Unit. let r = {x = ref 1, a = ref 0}  
              in 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 everytime `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 λ -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.

`setCounterClass =`

`λ r: CounterRep.`

`λ self: SetCounter ref.`

`{get = λ _ : Unit. !(r.x),`

`set = λ n: Nat. r.x := n,`

`inc = λ _ : Unit. (!self).set (succ ((!self).get unit))}`

The `self` parameter is a reference to a cell that contains the method of the current object.

Instantiation

- To create a counter, we first create a dummy counter and then subsequently set it:

```
dummySetCounter =  
  { get = λ _ : Unit. 0,  
    set = λ n:Nat. unit,  
    inc = λ _ : Unit. unit}'
```

```
newSetCounter =  
  λ _ : 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?

Subtyping References

- Try covariance:

$$\frac{\sigma <: \tau}{\sigma \text{ ref } <: \tau \text{ ref}}$$

- Assume $\sigma <: \tau$
 - The following holds:
 - $x : \tau, y : \sigma \text{ ref}, f : \sigma \rightarrow \text{int} \vdash y := x ; f (! y)$
 - Unsound: f is called on a τ but is defined only on σ
 - If we want covariance of references we can recover type safety with a runtime check for each $y := x$ assignment
 - The actual type of x matches the actual type of y
 - This is not a particularly good design.
 - Note: Java has covariant arrays
-

Subtyping References

Another approach (contravariance):

$$\frac{\sigma <: \tau}{\tau \text{ ref} <: \sigma \text{ ref}}$$

- Assume $\sigma <: \tau$
 - The following holds:
 - $x : \tau, y : \tau \text{ ref}, f : \sigma \rightarrow \text{int} \vdash y := x; f (!y)$
 - Unsound: f is called on a τ but is defined only on σ
 - references can be used in two ways:
 - for reading, context expects a value of type σ but the reference yields a value of type τ then we need $\tau <: \sigma$.
 - for writing, the new value provided by the context will have type σ . If the actual type of the reference is $\text{ref } \tau$, then this value may be read and used as a τ . This will only be safe if $\sigma <: \tau$.
-

Solution

- References should be type invariantly:
 - No subtyping for references (unless there are runtime checks)
 - Arrays (which are implemented using updates) should be typed invariantly.
 - Similarly, mutable records should also be invariant.
-

Refinements

- A value of type $\text{ref } \tau$ can be used in two different ways: as a source for reading and a sink for writing:
 - Split $\text{ref } \tau$ into three parts:
 - source τ : reference cell with "read" capability
 - sink τ : reference cell with "write" capability
 - $\text{ref } \tau$: cell with both capabilities
-

Modified Typing Rules

$$\frac{\Gamma, \Sigma \vdash e_1 : \text{source } \tau_1}{\Gamma, \Sigma \vdash !e_1 : \tau_1}$$
$$\Gamma, \Sigma \vdash !e_1 : \tau_1$$
$$\frac{\Gamma, \Sigma \vdash e_1 : \text{sink } \tau \quad \Gamma, \Sigma \vdash e_2 : \tau}{\Gamma, \Sigma \vdash e_1 := e_2 : \text{Unit}}$$
$$\Gamma, \Sigma \vdash e_1 := e_2 : \text{Unit}$$

Subtyping Rules

$$\sigma <: \tau$$

$$\text{source } \sigma <: \text{source } \tau$$

covariant on reads

$$\tau <: \sigma$$

$$\text{sink } \sigma <: \text{sink } \tau$$

contravariant on writes

$$\text{ref } \tau <: \text{source } \tau$$
$$\text{ref } \tau <: \text{sink } \tau$$

refs can be "downgraded"

to source or sink

Revisiting our example

- Use Source instead of Ref in the type of self:
 - pass the capability to read the method pointer, not the capability to write to it
 - permits covariant subtyping

Example

```
setCounterClass =  
  λ r: CounterRep.  
    λ self: Source SetCounter.  
      { get = λ _: Unit. !(r.x),  
        set = λ n: Nat. r.x := n,  
        inc = λ _:Unit. (!self).set (succ ((!self).get unit)) }
```
