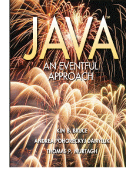
Semantics of Inheritance, revisited -- Gracefully

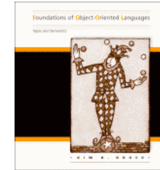
Kim Bruce
Pomona College

Background

- Live in two worlds:
  Teaching novices



Research in PL's



- Meld in designing Grace.

My Teaching Approach

- Start with objects -- concrete
- When want more than one, create a class
- Inheritance (from library classes) used early by students.
  - They design classes to inherit from much later.
- Like ideas in Scala, but consider it too complex for novices (and some parts too complex for me!)

Design Principles

- Steele's OOPSLA keynote on language design
  - Not too big, not too small
- Clean concepts more important than encoding everything from very small set of concepts
- In teaching:
  - Hide complexity until students can handle.
  - Use libraries to make programming more interesting.

Modeling Objects

- Semantics specified by Cook & Palsberg, as well as Cardelli, Kamin, and Reddy
- Cook *et al* provided typed model of objects and inheritance, while Bruce/Pierce & Turner provided different extensions supporting instance variables.
- Has anyone discussed the semantics of constructors?

Modeling Objects

- Objects consist of (shared) methods plus instance variables. Intuitively:

```
Obj = IV × (IV → Meth)
```

- Doesn't preserve subtypes under inheritance. Suppose `IV'` <: `IV`:

$$IV' \times (IV' \to Meth) \not<: IV \times (IV \to Meth)$$

- Solve with existential types

Modeling

- Type of objects: `μMT. ∃Y. Y × (Y → Meth(MT))` where
  - Y is type of record of instance variables
  - Meth is type of method suite
- If `obj` is an object, then `obj.m(x)` becomes:

```
open obj as <Rep, <iv, meth>>
                        in meth(iv).m(x)
```

Existentials for Information Hiding

- Advantage of using existentials for OO programs is can interchangeably use objects with same type but different representations.
  - Type only depends on method signatures
  - Ex.: Can mix cartesian and polar points in program.

Object Creation w/Classes

- Classes represent extensible object factories.
  - But not types!!!
  - class C.new(...) -> CType {
    def statVal : Tp1= ...
    var x: Tp2 := ...
    method m(...)->RT {...}
    doSomething // executable code
    }

    No separate constructor code

    class D.new(...) -> DType {
    inherits C.new(...)
    ... }

    All types optional

Semantics of classes

class C.new(...) {
  var iv_i := v_i
  method m_j (...){ $body_j$(self ...) }
}

Interpreted as pair:
  ({$iv_i = v_i$},
   λself. {$m_j = body_j$(self ...)}

- Create object via fixed point construction where allocate space for instance vbles & initialize with v_i.
- Assumption that instance vbles not refer to self.

Semantics of subclasses

class D.new(...) {
  inherits C.new(...)
  var iv_n := v_n *// new*
  method m_k (...){ $body_k$(self ...) } *// new or override*
}

Just add or replace instance vbles and methods from superclass.

My Graphics Library

- Applications need to register as a listener on their canvas to receive notification of mouse clicks
  - canvas.doSetUp(self)
- When code is in abstract superclass, had problems.
  - When inherited code executed in constructing subclass, used superclass self rather than subclass self.
  - Had to invoke explicitly in code of subclass
  - Clearly wrong!

Real World Object Creation

- What about initialization code?
  - What is meaning of self when run super init code?
  - In model, replace inst. vble values by init code (which may call self)
    - Need to be able to initialize constants in this code.
  - Extract implicit initialization code from class body & then:
    - create new object, then run initialization code:
      do superclass initialization (with new self),
      do new subclass initialization.

More Problems

- What about fields whose values are closures?
  - Can they reference self?
  - If so, what happens when they are inherited?

Object Calculus

- What is inheritance on objects?
  - Delegation or prototype
    - Currently Grace uses prototypes, but ...
  - Can emulate classes with objects
    - In reverse, object expression just creation of objects from anonymous classes.

Objects Emulating Classes

- Factory objects: methods return a new object
- Abadi-Cardelli have slightly more complex model to emulate classes
- Class object has "new" method plus fields for each method.
  - Fields for methods are closures taking self as parameter
  - "new" method sets methods of new object from fields

Objects representing Classes

- ```
  def Cclass = object {
      method new(...) {
          object{
              method mi(...) = outer.mi'(self,...) ..
          }
      def mi' = {sf,... -> bodymi(sf,...)}   // no use of self!
  }
  ```

Subclass

- For D to extend C:

```
def Dclass = object {
    method new(...) {
        object{
            method mi(...) {outer.mi'(self,...)}
            method n (...) {outer.n'(self,...)}  // new method
        }
    def mi' =  {sf,... -> c.mi'(sf,...)}
    def n' = {sf,... -> bodyn(sf,...)}    // overriding easy too
}
```

Types not in subtype relation! -- correct!
```

Object Inheritance in Grace

- An example:
  oc = object {...}
  od = object{ inherits objCreatorExp
  ... }
- When objCreatorExp is executed, what value is used for self in initialization? (If clone, no self!)
  - Should be self of extended object.

Conclusion

- Grace design nearly complete
- Syntax and semantics largely agreed upon
  - Some corner cases tricky
- Should we care if more complex concepts definable in terms of simpler???

Questions?

Bye!