Object-Oriented Languages, Fixed Points, and Systems of Objects

Kim Bruce
Williams College
on leave at U. of California at Santa Cruz

Based on work with Leaf Petersen, Joe Vanderwaart, and Nate Foster

FOOL History

- FOOL 1 - Stanford University in October 1993.
- FOOL 2 - Paris in July 1994.
  - *"On binary methods" w/ 8 co-authors*
- FOOL 3 - New Brunswick, New Jersey on July 1996.
- FOOL 4 - Paris in January 1997 w/ POPL

Outline

- History of Semantics of OO Languages
- Objects, Classes, and Fixed Points
- Problems When No Fixed Points
- Mutually Recursive Fixed Points

Restrict attention to class-based languages.

Early Semantics

- Cardelli
  - Objects as records of functions
- Cook & Palsberg, Reddy
  - Objects as fixed points of records
- Abel group
  - Typed semantics of objects and classes using fixed points *(no instance variables)*
- Kamin (untyped), Mitchell (typed)
  - Operational semantics using self-application

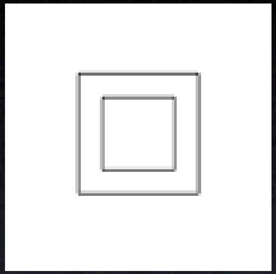
More early semantics

- Pierce, Bruce semantics with instance vbles
  - Both used existentials, differed in fixed points of types
- Abadi & Cardelli
  - operational semantics of object calculus
  - with Viswanathan, using fixed points on types
    - allows method updates in objects
- See “Comparing object encodings” by Bruce, Cardelli, & Pierce.

Fixed-point notation

- Let F: D → D
- Write Rec(d).F(d) for the "least" fixed point of F.
  - I.e., if a = Rec(d).F(d) then a = F(a).
- Say F is a generator of the fixed point
- We will interpret "this" or "self" in object-oriented languages as a fixed point.

A simple example



```
public class Squares {
    private FramedRect outer, inner;

    public Squares(Location upleft, int size,
        DrawingCanvas canvas){...}

    public void move(int dx, int dy) {
        this.outer.move(dx,dy);
        this.inner.move(dx,dy);
    }

    public void moveTo(int x, int y) {
        this.move(x - this.outer.getX(),  y - this.outer.getY());
    }
}
```

Understanding Objects

First naive view of objects:

```
⟦new Squares(...)⟧ =
    ({ outer = ref ...,              // no mention of this
       inner = ref ... },
     { move = fun(dx,dy). this.outer.move(...)...,
       moveTo = fun(x,y). this.move(...) })

where this = ({ outer = ...},
              { move = fun(dx,dy).
                    this.outer.move(...)...})
```

Understanding Objects

Equivalently:

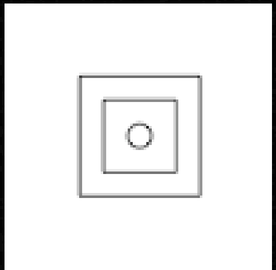
```
⟦new Squares(...)⟧ = Rec(this).
           ({ outer = ref ..., // no mention of this
              inner = ref ... },
            { move = fun(dx,dy). this.outer.move(...)...,
              moveTo = fun(x,y). this.move(...) })
```

Defines mutually recursive methods

Classes are Generators

- Classes serve many roles:
  - Types
  - Generators of new objects
- *Extensible* via subclasses to form new generators

Subclass



```
public class OvalSquares extends Squares {
    private FramedOval center;

    public OvalSquares(Location upleft,
            int size, DrawingCanvas canvas) {
        super(upleft, size, canvas);
        this.center = new FramedOval(...);
    }
    public void move(int dx, int dy) {
        super.move(dx, dy);  // old move
        this.center.move(dx, dy);
    }
}
```

What happens to moveTo?

Classes Are Generators of Fixed Points

- Meaning of *this* is not bound in classes
- Semantics of moveTo changes (indirectly) in OvalSquares:
  - someSquares = Rec(*this*). SQ(*this*)
  - someOvalSquares = Rec(*this'*). OSQ(*this'*) where OSQ extends SQ.
- Objects formed as *fixed points* of SQ and OSQ.

Semantics of Classes?

Methods must be meaningful in all possible subclasses.

$[\![$`class(i:I,m:M)`$]\!] =$

$$\Lambda\, M' <: [\![\mathtt{M}]\!].\ \Lambda\, IR' <: [\![\mathtt{I}^{ref}]\!].$$

$$(\,[\![\mathtt{i}]\!]\ ,\ \lambda(this : IR' \times (IR' \rightarrow M')).\ [\![\mathtt{m}]\!]\,)$$

initial values of instance variables

methods

Semantics of Objects

$[\![\texttt{new Squares(...)}]\!] =$

$(\{\ \mathit{outer} = \mathit{ref}\ ...,\ \mathit{inner} = \mathit{ref}\ ...\ \},$

$\mathrm{Rec}(\mathit{fm} : [\![\texttt{I}^{\mathit{ref}}]\!] \rightarrow [\![\texttt{M}]\!]).$

$\lambda(\mathit{inst} : [\![\texttt{I}^{\mathit{ref}}]\!]).$

$\{\ \mathit{move} = \mathit{fun}(dx,dy).\ \mathit{inst}.\mathit{outer}...,$

$\mathit{moveTo} = \mathit{fun}(x,y).\langle \mathit{inst},\mathit{fm} \rangle.\mathit{move}(...)\ \})$

Method suite can be shared between objects of same type.

Sending Messages

Suppose $[\![\texttt{obj}]\!] = \langle \texttt{i}, \texttt{fm} \rangle$

then $[\![\texttt{obj.p(...)}]\!] = \mathit{fm}(i).p(\ldots)$

In objects, methods fixed -- parameterized by suite of instance variables, not this.

Summary of Semantics

- *Fixed points* are key to understanding O-O languages.
- Classes are *extensible generators* of fixed points.
- Objects are formed as *fixed points*.

Type of Class

$$[\![\texttt{class(i:I,m:M)}]\!] =$$

$$\Lambda\, M' <: [\![\texttt{M}]\!].\, \Lambda\, IR' <: [\![\texttt{I}^{ref}]\!].$$

$$(\, [\![\texttt{i}]\!]\, ,\, \lambda(this : IR' \times (IR' \rightarrow M')).\ [\![\texttt{m}]\!]\,)$$

$$[\![\texttt{Class(I,M)}]\!] =$$

$$\forall\, M' <: [\![\texttt{M}]\!].\, \forall\, IR' <: [\![\texttt{I}^{ref}]\!].$$

$$[\![\texttt{I}]\!] \times (\underline{IR' \times (IR' \rightarrow M')} \rightarrow [\![\texttt{M}]\!])$$

$ThisClass_{int}$

The Type of "this"

Inside methods

this : ThisClass_{int}

where $[\![\text{ThisClass}_{int}]\!] = IR' \times (IR' \rightarrow M')$

When return this, hide instance variables:

this : ThisClass_{ext}

where $[\![\text{ThisClass}_{ext}]\!] = \exists X.\, X \times (X \rightarrow M')$

ThisClass as a Type

Can declare:

clone: () → @ThisClass

If obj has type @T, then value of obj can have type T, but not a subtype.

Subsumption: e: @T ⇒ e: T

Also examples with parameters of type ThisClass -- called *binary methods*

LOOJ Example

(Bruce & Foster, ECOOP '04)

```
public class Node {
   protected @ThisClass next;
   public Node(@ThisClass next) {
     this.next = next;
   }

   public void setNext(@ThisClass newNext) {
     this.next = next;
   }

   public @ThisClass getNext() {
     return next;
   }
}
```

binary method

Subclasses

```
public class DbleNode extends Node {
   protected @ThisClass prev;
   public DbleNode(@ThisClass next,
                   @ThisClass prev) {
     super(next);
     this.prev = prev;
   }
   public void setNext(@ThisClass newNext) {
     super.setNext(newNext);
     newNext.setPrev(this);
   }
   public void setPrev(@ThisClass newPrev) {
     this.prev = newPrev;
   }
   ...
}
```

Type-Checking

- Can only send binary message to object if know its exact type - @C.
- If o: @C and m: U then o.m: U[C/ThisClass]
- If o: C then o.m: U[C/@ThisClass, ThisClass]
  - If m: @Node then m.clone(): @Node
  - If n: Node then n.clone(): Node
  - m.setNext(m) *legal*, n.setNext(m) *illegal*

Type of Objects

If get object from

```
class C{i:I(ThisClass),
        m:M(ThisClass)}
```

Then meaning of type of objects is:

$$[\![C]\!] = \mathit{Rec}(\mathit{ThisClass}).\ \exists X.\ X \times (X \to M(\mathit{ThisClass}))$$

Type-Checking Classes

- Type-check modularly.
- Type-check methods of class C under assumptions that hold in all extensions!
  - this : ThisClass
  - ThisClass extends C

Can prove soundness of type system. (ECOOP '04)

F-bounded Polymorphism

- F-bounded polymorphism introduced by Mitchell and the Abel group (similar to other proposals). *Now in Java 5.*
- Very expressive, but not work smoothly with inheritance hierarchy.

F-bounded Polymorphism

```
abstract class ComparableF<T> {
   boolean lessThan(T other); }

class BinTree<T extends ComparableF<T>>
 {...}

class Point extends ComparableF<Point>{
   boolean lessThan(Point other){...} }

class ColorPoint extends Point {...}
```

Is `BinTree<ColorPoint>` *legal?*

In some situations need `T = ComparableF<T>`

Fixed w/ ThisClass

```
abstract class Comparable {
   boolean lessThan(ThisClass other); }

class BinTree<T extends Comparable>
 {...}

class Point implements Comparable{
   boolean lessThan(ThisClass other){...} }

class ColorPoint extends Point {...}
```

`BinTree<ColorPoint>` *is legal!*

Interacting Systems

- Expression problem
  - *See solution in §6 of my WOOD 2003 paper.*
  - *Scala solution later today*
- Subject-Observer pattern

Virtual Classes

```
class Observer {
  typedef SType as Subject;
  typedef EType as Event;
  void notify (SType subj, EType e) {…};
}

class Subject {
  typedef OType as Observer;
  typedef EType as Event;
  OType[] observers;        ...
  void notifyObservers(EType e) {
    for (obs:observers)
      obs.notify(this,e);
  }
}
```

Specializing Virtual Classes

```
class MenuObserver extends Observer {
  typedef SType as MenuSubject;
  typedef EType as MenuEvent;
  void notify (SType subj, EType e) {
    … subj.getSelectedLabel() …};
}

class MenuSubject extends Subject {
  typedef OType as MenuObserver;
  typedef EType as MenuEvent;
  void getSelectedLabel(){…}
    …
}
```

Mutually Recursive Groups

```
group SubjectObserver{
  class Observer {
    void notify (@ThisTG.Subject subj,
                 @ThisTG.EventType e) {…};
  }
  class Subject {
    void notifyObservers(@ThisTG.EventType e) {
      for (obs:observers)
        obs.notify(this,e);
    }
  }
  class Event {…}
}
```

Extending Groups

```
group MenuSubObs extends SubjectObserver{
  class Observer {
    void notify (@ThisTG.Subject subj,
                @ThisTG.EventType e) {
        … subj.getSelectedLabel()…};
  }
  class Subject {
    String[] labels;
    String getSelectedLabel() {…};
  }
  …
}
```

Semantics of Groups

If group TG {C_1 = Class(…,M_1(ThisTG)), …,

C_n = Class(…,M_n(ThisTG)) }

then types corresponding to `TG` given by:

$$Rec(ThisTG).\{C_1 = \exists X.\, X \times (X \rightarrow M_1(ThisTG))\ldots,$$

$$C_n = \exists X.\, X \times (X \rightarrow M_n(ThisTG))\ \}$$

Type-Checking Rules

- ... similar to as before.
  - Need exact types for receiver of methods with ThisTG in parameter positions.
  - Type-check under weak assumptions on ThisTG
  - Proof of type-safety similar
    - *done for extension of LOOM, not LOOJ*

Even More ...

- Groups can preserve relations between classes
  - If class A extends B in group G, and G' extends G, then A also extends B in G'

Example

Borrowed from Jiazzi

```
group GUIComponents {
   class Component {...}
   class Button extends ThisTG.Component {...}
   class Window extends ThisTG.Component {
      void addComp(@ThisTG.Component c){...}
   }
}
```

Extending ...

```
group ColorGUIComponents
                          extends Components {
   class Component {
      Color currentColor;
      void setColor(Color clr){...}
   }
}
```

All extensions of Component gain Color.

Type error if conflicts!

Polymorphism Works

```
@ColorComponents.Window clrWindow;
@ColorComponents.Button clrButton;

clrWindow.addComponent(clrButton);
      // because Button still extends Window

<G extends GUIComponents>
  void doGUIStuff (@G.Button button,
                   @G.Window window) {
      ... window.addComponent(button)... }
```

Type-checking rules ensure safety!

ThisTG vs. ThisType

- ThisType as fixed point:
  - Works well, but complications because of exact types and binary methods.
- ThisTG as mutually recursive fixed point
  - Works better, especially if default is exact group.
  - Generalization may be more fundamental than original.

Related Work

- Beta's Virtual Types and proposed Java extensions by Thorup and Torgersen
- Bruce & Vanderwaart '99
  - w/weaker type system
- Family Polymorphism - Erik Ernst
- Scala - Odersky et al - dependent types
- Nested Inheritance - Nystrom, Chong, & Myers *(restricted to exact types)*

More Related Work

- Hand-rolled fixed points:
  - Units & Mixins - Findler and Flatt
  - Jiazzi (Java extension) - McDirmid, Flatt, and Hsieh
  - Collaborations - Smaragdakis and Batory
- OCAML - Remy & Vouillon
  - Type inference and row types

Questions?