Fixing the Meaning of Object-Oriented Languages

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

What makes O-O Languages Special?

- Objects encapsulate state & methods
- Subtyping
- Inheritance
- Self-reference
  - *Provides dynamic method invocation with more power!*

Luca Cardelli: "If there is no 'self' in a language then it is not object-oriented."

What's the Big Deal?

- Are objects more than records with function components?
- What provides real power?
- How can semantics and type theory help?
- Focus on class-based O-O languages like Smalltalk, Eiffel, & Java
  - Multi-method languages are quite different

Defining a Class

```
public class Squares {
  private FramedRect outer, inner;
  public Squares(Location upleft, int size,
                 DrawingCanvas canvas){...}
  public void move(int dx, int dy) {
    outer.move(dx,dy);
    inner.move(dx,dy);
  }
  public void moveTo(int x, int y) {
    this.move(x - outer.getX(),
              y - outer.getY());
  }
}
```


Instance Variables

```
public class Squares {
  private FramedRect outer, inner;
  public Squares(Location upleft, int size,
                 DrawingCanvas canvas){...}
  public void move(int dx, int dy) {
    outer.move(dx,dy);
    inner.move(dx,dy);
  }
  public void moveTo(int x, int y) {
    this.move(x - outer.getX(),
              y - outer.getY());
  }
}
```

Constructor

```
public class Squares {
  private FramedRect outer, inner;
  public Squares(Location upleft, int size,
                 DrawingCanvas canvas){...}
  public void move(int dx, int dy) {
    outer.move(dx,dy);
    inner.move(dx,dy);
  }
  public void moveTo(int x, int y) {
    this.move(x - outer.getX(),
              y - outer.getY());
  }
}
```


Methods

```
public class Squares {
  private FramedRect outer, inner;
  public Squares(Location upleft, int size,
                 DrawingCanvas canvas){...}
  public void move(int dx, int dy) {
    outer.move(dx,dy);
    inner.move(dx,dy);
  }
  public void moveTo(int x, int y) {
    this.move(x - outer.getX(),
              y - outer.getY());
  }
}
```


Creating & Using Objects

```
Squares fst = new Squares(corner,10,canvas);
Squares snd = new Squares(middle,40,canvas);
// objects are references
fst.moveTo(20,30);
snd = fst;     // snd & fst refer to same object
snd.move(30,50);
```


Objects Are Fixed Points

First naive view of objects:

[[`new Squares(...)`]] =

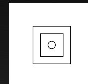
μ *this*.({ *outer* = ..., // *no mention of this*
inner = ... } ×
{ *move* = *fun(dx,dy)*. *this.outer.move()...*,
moveTo = *fun(x,y)*. *this.move(...)* })

Defines mutually recursive methods.

Classes Are Generators

- Classes serve many roles:
  - Types
  - Generators of new objects
  - Extensible 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);
      center = new FramedOval(...);
   }
   public void move(int dx, int dy) {
      super.move(dx, dy);  // old move
      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
- class Squares = λ *this*. SQ(*this*)
- class OvalSquares = λ *this*. OSQ(*this*) where OSQ extends SQ.
- Objects formed as *fixed points* of SQ and OSQ.

Objects From Subclasses

- `sq = new Squares(...);`
  - *sq = μ this. SQ(this)*
- `osq = new OvalSquares(...);`
  - *osq = μ this'. OSQ(this') // meaning of this changed!*
  - *where super = SQ(this') // uses new this' in body*

Subtyping

- T <: U *iff* any object of type T can *masquerade* as object of type U
- More formally, *subsumption* rule:

  T <: U & o : T ⇒ o : U
- Java Interfaces & extension

Subtyping Immutable Record Types

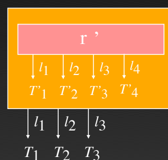
Records without field update: only operation is extracting field: ... s.filling ...

{bread: BreadTp; filling: CheeseTp; sauce: SauceTp}
<:
{ bread: BreadType; filling: FoodType }

iff

CheeseTp <: FoodType

Subtyping Immutable Record Types



$$\{ l_i : T'_i \}_{1 \le i \le n} <: \{ l_i : T_i \}_{1 \le i \le k}$$

iff

$k \le n$ *and for all* $1 \le i \le k$, $T'_i <: T_i$.

Subtyping Function Types

If f : S → T and s : S then f (s) : T

When is S' → T' <: S → T ?

If f' : S' → T' and s : S, need f' (s) : T.

Subtyping Function Types

$$S' \rightarrow T' <: S \rightarrow T$$

iff

$$S <: S' \text{ and } T' <: T.$$

Contravariant for parameter types.
Covariant for result types.

Subtyping Object Types

$$\text{ObjType } \{ m_i : T'_i \}_{1 \le i \le n} <: \text{ObjType } \{ m_i : T_i \}_{1 \le i \le k}$$

iff

$k \le n$ *and for all* $1 \le i \le k$, $T'_i <: T_i$.

assuming methods not updatable at run-time!

Method parameters can vary contravariantly, return types covariantly.

Restriction on Subclass Changes

- Java 1.4 doesn't allow any changes to types of methods in subclass.
- C++ & Java 5 allow covariant changes to return types.
- Suppose you don't care if subclass gives a subtype. Do you still need restrictions?
  - *In Smalltalk, subclass and subtype hierarchies sometimes reversed.*


```
class Example {
      :
    void m(...) {... this.n(s) ...}
    T n(S x) {...}
}

class SubExample extends Example {
    T' n(S' x) {...}
    void newMeth(...) {...}
}
```

What is relationship of new type of n to old if want type safety?

Restriction on Subclass Changes

- Method type in subclass must be subtype of method type in superclass for safety:
  - Covariant change allowed in return type
  - Contravariant change in parameter type

Semantics of Classes?

- Methods must be meaningful in all possible subclasses.

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

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

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

initial values of instance variables — *methods*

Semantics of Objects

[[`new Squares(...)`]] =

{ *outer = ref ..., inner = ref ...* }

×

μ(*fm* : [[$\texttt{I}^{ref}$]] → [[`M`]]).
  λ(*inst* : [[$\texttt{I}^{ref}$]]).
    { *move = fun(dx,dy). inst.outer...,*
      moveTo = fun(x,y). ⟨*inst,fm*⟩ *.move(...)* }

Method suite can be shared between objects of same type.

Sending Messages

[[`obj.p(...)`]] = *fm(i).p(...)*

where [[`obj`]] = ⟨`i`, `fm`⟩

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

Summary

- *Fixed points* are key to understanding O-O languages.
- Classes are *extensible generators* of fixed points.
- Subtyping explains restrictions on subclasses
  - *Even though subtyping distinct concept.*

What is Type of "this"?

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

∀*M'* <: [[`M`]]. ∀*IR'* <: [[$\texttt{I}^{ref}$]].

[[`i`]] × λ(*this : IR' × (IR' → M')*). [[`m`]]

How do we type function returning *this*?

Finding a Type for "this"

- Defining this in Class(i : I, m : M)
  - this : IR' × (IR' → M') where M' <: [[`M`]] , IR' <: [[$\texttt{I}^{ref}$]]
- When return this, hide instance variables
  - this : ∃X. X × (X → M') = Obj(M')
  - but notice Obj(M') <: Obj(M), thus this : Obj(M)
    - where Obj(M) is type of objects of the class being defined.

Losing Information

- But now we have lost information.
- If C has method m() with body return this, then C m().
- If D extends C, then inherited m still has type C m().
- But d.m() really returns value of type D!

Introduce Type of "this"

- ThisClass is type of "this"
- In previous example, declare:
  - m: () → ThisClass
- Also examples with parameters of type ThisClass -- called *binary methods*

Binary Methods

```
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;
  }
}
```


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 ThisClass setPrev(ThisClass newPrev) {
    this.prev = newPrev;
  }
  ...
}
```


But ....

There are type safety problems! Define:

```
public void breakIt(Node n1, Node n2) {
      n1.setNext(n2);
 }
```

Let n be a Node and dn be a DbleNode:

```
breakIt(dn,n);
```

Blows up when `dn.setNext(n)` is executed!

Need to rule out this code using type system.

Fix Type-Safety

- Can only send binary message to object if know its exact type.
- If obj has type @T, then value of obj has type T, but not a subtype.

Using Exact Types

Rewrite Node and DbleNode to replace all occurrences of ThisClass by @ThisClass:

```
public void breakIt(@Node n1, @Node n2) {
  n1.setNext(n2);
}
```

More generally w/ type parameters could write:

```
public <N extends Node>
          void breakIt(@N n1, @N n2) {
  n1.setNext(n2);
}
```

Meaning of ThisClass

- ThisClass is type of this.
- In semantics:
  - Node = μThisClass.Nd(ThisClass)
  - DbleNode = μThisClass'.DbNd(ThisClass');
- Thus
  - ThisClass = Nd(ThisClass) = Node
  - ThisClass' = DbNd(ThisClass') = DoubleNode

Type-Checking

- Wish to type-check modularly.
  - no repeated type-checking of inherited methods
- Type-check methods of class C under assumptions that hold in all extensions!

this : ThisClass
ThisClass extends C

Can prove soundness of type system.

Verification

- Verification can benefit from similar assumptions on this and ThisClass
- Alternative: Verify under closed-world assumption and copy inherited methods down.
  - *Compositional approach is better!*

There Is Much More ...

- Gets *much* more interesting when:
  - Allow type parameters (e.g., GJ)
  - Introduce ThisType as interface of this.
    - LOOJ extension of GJ (Java 5)
  - Allow mutually recursive ThisClasses in *class groups*.
  - Allows types and classes to be refined simultaneously in type-safe fashion.

Subject-Observer Example

```
SubjObsGrp = ClassGroup {
   class MySubject {
     void addObserver(MyObserver myObs) {...};
     void notifyObservers: proc(MyEvent myEvt) {...}; }

   class MyObserver {
     void notify(MySubject mySubj, MyEvent myEvt) {...}; }

   class MyEvent {…}
}
```


Extending Class Groups

```
ChoiceSubjObsGrp = ClassGroup extends SubjObsGrp {
   class extends MySubject {
      String getItem() {...}; }

   class extends MyEvent {
      String getItem() {...}; }

   class extends MyObserver {
      void notify(MySubject mySubj, MyEvent myEvt) {
         ... myEvt.getItem() ...}; }
}
```

Summary

- Fixed points play key role in OO languages.
- Classes are generators of fixed points:
  - cl = λ *this*. *CL*(*this*)
  - obj = μ *this*. *CL*(*this*)
- Types similar:
  - Class = λ *ThisClass*. *CLType*(*ThisClass*)
  - ObjTp = μ *ThisClass*. *CLType*(*ThisClass*)

Summary (*continued*)

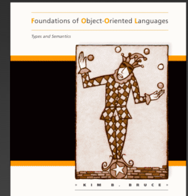
- Type-checking must take into consideration future subclasses.
- Verification should also plan for subclasses.

Summary (*continued*)

- Interdependent groups of objects common.
- Need to subclass simultaneously
  - Not supported in current languages
  - Mutually recursive ThisClasses provide opportunities for type-safe specialization.

References

- LOOJ: Weaving LOOM into Java, ECOOP 2004.
- Some Challenging Typing Issues in Object-Oriented Languages, ENTCS 82, #8, 2003.
- Foundations of Object-Oriented Languages: Types and Semantics, MIT Press, 2002.



http://www.cs.williams.edu/~kim

Questions?