What Semantics Can Teach Functional Programmers About Object-Oriented Languages

Kim Bruce
Williams College
O-O Languages Hot

- Seem to be a great improvement over procedural languages
  - Objects encapsulate state & methods
  - Subtyping
  - Inheritance
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
public class Squares {
    private FilledRect 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());
    }
}
```java
public class Squares {
    private FilledRect 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());
    }
}
```
public class Squares {
    private FilledRect 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());
    }
}
public class Squares {
    private FilledRect 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
fst.move(30,50);
Objects Are Fixed Points

First naive view of objects:

\[
[[ \text{new Squares}(\ldots) ]] = \\
\mu \ this.\{ \text{outer} = \ldots, \quad // \text{no mention of this} \\
\text{inner} = \ldots \} \times \\
\{ \text{move} = \text{fun}(dx,dy). \ this.\text{outer}..., \\
\text{moveTo} = \text{fun}(x,y). \ this.\text{move}(\ldots) \}
\]

Defines mutually recursive methods.
Classes Are Generators

- Classes serve many roles:
  - Types
  - Generate new objects
  - Extensible to form new generators
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);
    }
}
Classes Are Generators of Fixed Points

• Meaning of *this* is not bound in classes
  - Semantics of moveTo changes (indirectly) in OvalSquares

• Squares = SQ(this)

• OvalSquares = 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

• Related to signature matching in ML and type classes in Haskell

• $T <: U$ iff any object of type $T$ can masquerade as object of type $U$

• More formally, *subsumption* rule:

  $$T <: U \& o : T \implies o : U$$

• 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

\[
\{ l_i : T' \}_{1 \leq i \leq n} \lll \{ l_i : T \}_{1 \leq i \leq k} \quad \text{iff} \\
\quad k \leq n \text{ and for all } 1 \leq i \leq k, \quad T'_i \lll T_i.
\]
If \( f : S \rightarrow T \) and \( s : S \) then \( f (s) : T \)

When is \( S' \rightarrow T' \) <: \( S \rightarrow T \)?

If \( f' : S' \rightarrow T' \) and \( s : S \), need \( f' (s) : T \).
Subtyping Function Types

\[ S \rightarrow S' \rightarrow f' \rightarrow T' \rightarrow T \]

\[ S' \rightarrow T' <: S \rightarrow T \]

iff

\[ S <: S' \text{ and } T' <: T. \]

**Contravariant** for parameter types.

**Covariant** for result types.
Variables can be *suppliers* & *receivers* of values.

\[ x := x + 1 \]

If \( x \) is a vble of type \( T \), write \( x : \text{ref } T \).

**When is \( \text{ref } T' <: \text{ref } T \)?**

To replace variable \( x : \text{ref } T \) by \( x' : \text{ref } T' \) in:

*expression*: \( \ldots x \ldots \)

Need \( T' <: T \).

*assignment*: \( x := e \) where \( e : T \).

Need \( T <: T' \).
Subtyping Reference Types

Supplier: *covariant*; Receiver: *contravariant*

\[
\text{ref } T' <: \text{ ref } T \iff T' \approx T
\]
Subtyping Updatable Record Types

Updatable Records:

When is \( \{ l_i : T'_i \rangle_{1 \leq i \leq n} \prec \{ l_i : T_i } \rangle_{1 \leq i \leq k} \) ?

... \( r.l_i := e \) ...
Arrays:

If $S <: T$, is $\text{Array of } S <: \text{Array of } T$?

Java says yes, but ...

With few exceptions, for $F:\text{Types} \rightarrow \text{Types}$,

$S <: T \Rightarrow F(S) <: F(T)$. 
Subtyping Object Types

\[
\text{ObjType } \{ m_i : T'_i \}_{1 \leq i \leq n} \lessdot \text{ObjType } \{ m_i : T_i \}_{1 \leq i \leq k}
\]

iff

\[
k \leq n \text{ and for all } 1 \leq i \leq k, \quad T'_i \lessdot T_i.
\]

only if methods not updatable at run-time!

*Method parameter can vary contravariantly, return types covariantly.*
Restriction on Subclass Changes

- Java doesn’t allow any changes to types of methods in subclass.
- C++ allows 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 retain meaning in subclasses.

\[
[[\text{class}(i:I,m:M)]] = \forall M' \triangleleft [[M]]. \forall IR' \triangleleft [[I^\text{ref}]].

[[i]] \times \lambda (this : IR' \times (IR' \rightarrow M')).
\]

[[m]]
Semantics of Objects

\[
[[ \text{new } \text{Squares}(...) ]] = \\
\{ \text{outer} = \text{ref} ..., \text{inner} = \text{ref} ... \} \times \\
\mu(fm : \llbracket I^\text{ref} \rrbracket \to \llbracket M \rrbracket).
\]
\[
\lambda(\text{inst} : \llbracket I^\text{ref} \rrbracket).
\]
\[
\{ \text{move} = \text{fun}(dx,dy). \text{inst}.\text{outer}..., \\
\text{moveTo} = \text{fun}(x,y). \langle \text{inst},fm \rangle.\text{move}(...) \}
\]

Also information hiding with existential types - for correctness & type safety!
Sending Messages

\[
[[\text{obj}.\text{p}(\ldots)]] = \text{fm}(i).\text{p}(\ldots)
\]

where \([\text{obj}] = \langle i, \text{fm} \rangle\)

*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.
There Is Much More ...

- Gets *much* more interesting when:
  - Allow type parameters (e.g., GJ)
  - Allow type for this: **ThisType**
  - Consider weaker relations than subtyping
    - e.g., *matching*
Questions?

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