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

Defining a Class

```
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());
  }
}
```

Instance Variables

```
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());
  }
}
```

Constructor

```
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());
  }
}
```

Methods

```
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:

$[\![$ `new Squares(...)` $]\!]$ =

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

Defines mutually recursive methods.

Classes Are Generators

- Classes serve many roles:
  - Types
  - Generate 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);
   }
}
```

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:

$$T <: U \ \& \ o : T \quad \Rightarrow \quad o : U$$

- Java Interfaces & extension

Subtyping Immutable Record Types

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

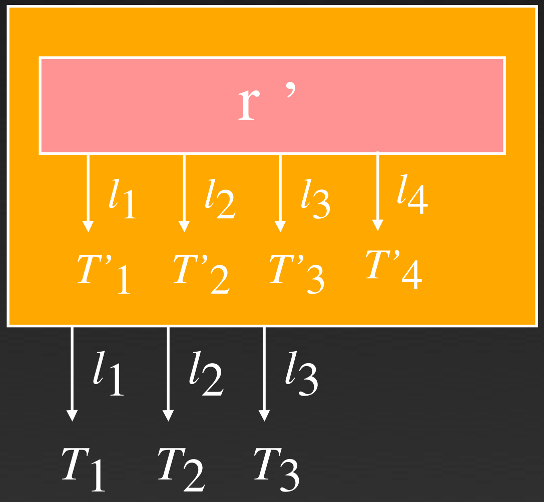
{bread: BreadTp; filling: CheeseTp; sauce: SauceTp}

<:

{ bread: BreadType; filling: FoodType }

iff CheeseTp <: FoodType

Subtyping Immutable Record Types



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

iff

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

Subtyping Function Types

If $f : S \to T$ and $s : S$ then $f(s) : T$

When is $S' \to T' <: S \to T$?

If $f' : S' \to T'$ and $s : S$, need $f'(s) : T$.

Subtyping Function Types

$$S' \to T' \;<:\; S \to T$$

iff

$S <: S'$ *and* $T' <: T$.

Contravariant for parameter types.
Covariant for result types.

Subtyping Reference Types

Variables can be *suppliers* & *receivers* of values.

x := x + 1

If x is a vble of type T, write x: ref T.

When is ref T’ <: ref T?

To replace variable x : ref T by x' : ref T' in:

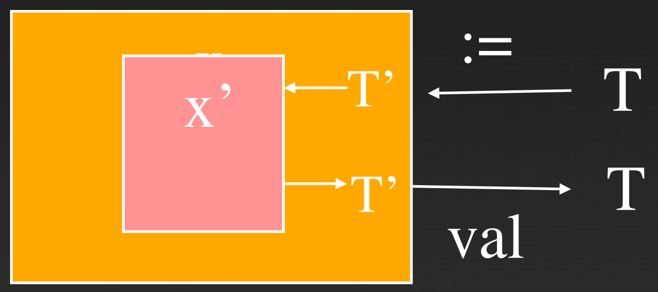
expression: … x …

Need T' <: T.

assignment: x := e where e:T.

Need T <: T'.

Subtyping Reference Types



Supplier: *covariant*; Receiver: *contravariant*

ref T' <: ref T *iff* T' ≈ T

Subtyping Updatable Record Types

Updatable Records:

When is $\{ l_i : T'_i \}_{1 \leq i \leq n}$ <: $\{ l_i : T_i \}_{1 \leq i \leq k}$?

… $r.l_i$:= e ...

Subtyping Array Types

Arrays:

If S <: T, *is* Array of S <: Array of T?

Java says yes, but ...

With few exceptions, for $F: Types \rightarrow Types$,

$S <: T \not\Rightarrow F(S) <: F(T)$.

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* $1 \le i \le k$, $T'_i <: T_i$.

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.

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

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

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

Semantics of Objects

$[\![$ `new Squares(...)` $]\!] =$

$\{$ *outer* = *ref* ..., *inner* = *ref* ... $\} \times$

$\mu(fm : [\![\, \texttt{I}^{ref} \,]\!] \rightarrow [\![\, \texttt{M} \,]\!]).$

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

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

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

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

Sending Messages

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

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

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

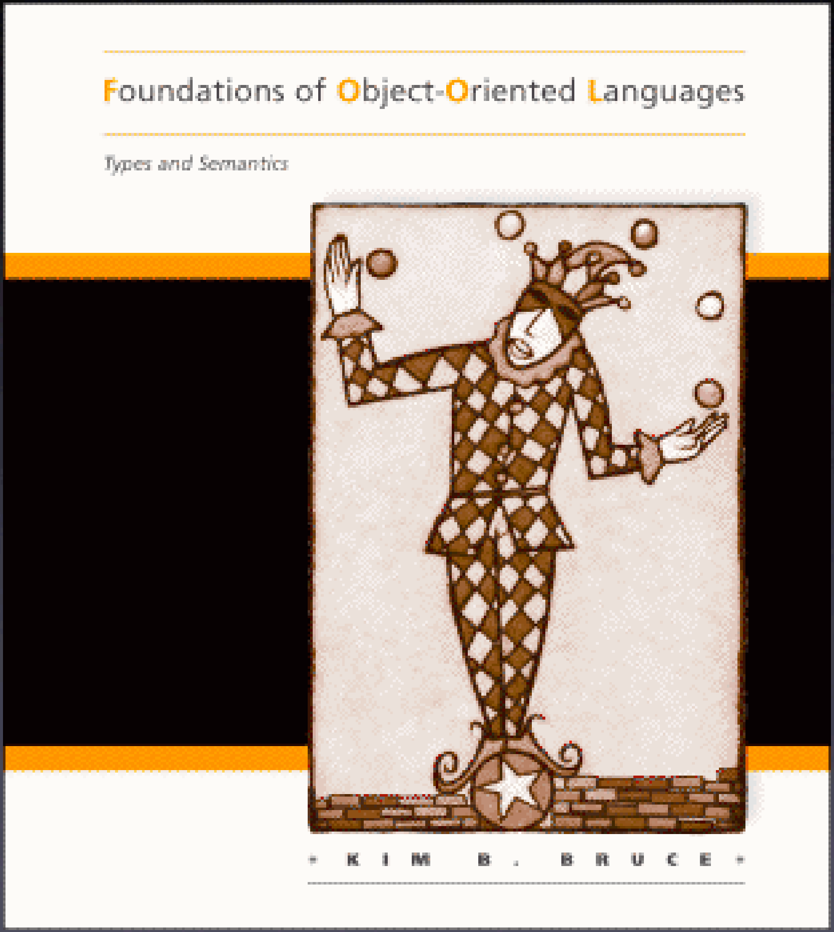
Summary

- *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