# Thoughts on Subtypes versus Inheritance

Kim Bruce
Williams College

# Is There a Real Difference?

- Differences between subtypes and subclasses (inheritance) recognized early:
  - Pierre America
  - Cook et al.
- Conflated in popular O-O languages.

# Subtyping

- S is a *subtype* of T (written S <: T) iff an object of S can "masquerade" as an object of T in any context

  iff

  for all m: U → V in T, there is an m: U' → V' in S such that V' <: V & U <: U'.

- Formally expressed via subsumption rule:

$$\frac{o : S \qquad S <: T}{o : T}$$

# *Thoughts on Subtypes*

- Key is knowing what interfaces an object satisfies.
- Subtype ordering & subsumption are a convenience: allow programmers to write small number of interfaces.
- Some contexts (parametric polymorphism) require name for fixed subtype, e.g., `class OrderList< T <: Comparable>`
- *Behavioral subtyping* is required to reason about programs.

# Inheritance

- Enables incremental changes to classes.
- Abstract out repeated code.
- Modify existing methods from superclass.
- Add new methods and instance variables.
- Modify behavior without breaking inherited methods.

# Type Changes in Subclasses

- Restrictions necessary to retain static type safety:

```
public class C {
  public V m(U arg) { … }
    public void n( ) { … this.m(anArg) … }
}

public class D extends C {
   public V′ m(U′ arg) { ... }
}
```

*Need V' <: V, U <: V' for type safety. I.e.,*

$U' \rightarrow V' <: U \rightarrow V$

# Subclasses Give Rise to Subtypes

- To retain type safety:
  - Types of methods in subclass must be subtype of method types in superclass.
  - Types of instance variables can't change in subclass.
- Unless hide methods, subclass always gives rise to subtype.
- Most type-safe O-O languages don't allow any changes to types of methods in subclass.

# Lose Precision of Types in Subclass

```
public class Node {
  private Node next;
  public Node getNext( ) { return next; }
  public void setNext(Node newNext) { … }
}

public class DbleNode extends Node {
  private DbleNode previous;
  public void setNext(DbleNode newNext ) { … }
  public DbleNode getPrev( ) { … }
  public void setPrev(DbleNode newPrev) { … }
}
```

*Also wrong type for inherited* next *&* getNext

# Regain Precision with MyType

```
public class Node {
  private MyType next;
  public MyType getNext( ) { return next; }
  public void setNext(MyType newNext) { … }
}

public class DbleNode extends Node {
  private MyType previous;
  public void setNext(MyType newNext ) { … }
  public MyType getPrev( ) { … }
  public void setPrev(MyType newPrev) { … }
}
```

*Correct types for inherited* next *&* getNext *in* DbleNode.

# Lose Subtyping

- DbleNode not subtype of Node:

```
void breakit(Node nd, Node nd') {
    nd.setNext(nd');
}
```

breakit(nd1,nd2) OK, if nd1, nd2 both of type Node, not if nd1 is DbleNode and nd2 is Node.

- Inheritance still useful (and type-safe!).

# Replace Subtyping by Matching

- Define S matches T (written S <# T) iff
  for all m: U→V in T there is an m: U→V in S
  *(ignoring implicit change in MyType).*
- typeof(DbleNode) <# typeOf(Node):
  - e.g., setNode: MyType → void in both Node & DbleNode.
- Types of subclass and superclass always match.

# How is Matching Used?

$$\frac{C \vdash S <\# \ \text{Interface}\{ m : T \}, \quad C, E \vdash o : S}{C, E \vdash o.m : T[\, S \,/\, \text{MyType} \,]}$$

*if MyType does not occur as parameter type in m.*

Need *exact types* for type safety with binary methods.
Write o: @S to mean o has type S, but not extension.

$$\frac{C \vdash S <\# \ \text{Interface}\{ m : T \}, \quad C, E \vdash o : @S}{C, E \vdash o.m : T[\, @S \,/\, @\text{MyType} \,][\, @S \,/\, \text{MyType} \,]}$$

# What Good is Matching?

- New rules replace subsumption rule!
  - Though LOOJ backward compatible with Java!
  - Just interpret "extends" as matching.
- Matching tells what messages can be sent to object.
- In most cases, that is sufficient.
- Can also use as bound for polymorphism:
  `class C<T extends Comparable> { … }`

# Are There More?

- Are there other relations as useful as subtyping and matching?
- If start w/ default of exact types then introduce hash types (#T) for values of type T or extension:

  $\#T = \exists\ t <\# T.\ t$

- Can think of subtyping same way.
- New wild card types in Java 1.5 also definable in terms of existential types:
  - $F\langle * \rangle = \exists\ t.\ F(t), \quad F\langle \text{-}T \rangle = \exists\ t :> T.\ F(t)$

# Mutually Recursive Types

```
interfaceGroup SubjObsGrp {
  interface MySubject {
    void addObserver(MyObserver obs);
    void notifyObservers(MyEvent evt);
  }

  interface MyObserver {
    void notify(MySubject subj, MyEvent evt);
  }

  interface MyEvent {…}
}
```

# Classes & Types

```
class SubjectClass
          implements SubjObsGrp.MySubject {
  MyObserver[ ] observers;   …

   void addObserver(MyObserver obs) { … }
   void notifyObservers(MyEvent evt) { … }
}

class ObserverClass
          implements SubjObsGrp.MyObserver {
  …
  void notify(MySubject subj, MyEvent evt)
  { … }
}

class EventClass implements ….MyEvent { … }
```

# Extending Mutually Recursive Types

```
interfaceGroup ChoiceSubjObsGrp
                    extends SubjObsGrp {
  interface MySubject {
    String getItem();
  }

  interface MyEvent {
    String getItem();
  }
}
```

# Extending Classes

```
class ExtSubjectClass extends SubjectClass
      implements ExtSubjObsGrp.MySubject {
   String getItem(){ … }
   void notifyObservers(MyEvent evt) { … }
}

class ExtObserverClass extends ObserverClass
      implements ExtSubjObsGrp.MyObserver {
  void notify(MySubject subj, MyEvent evt)
   { … evt.getItem() … }
}
class ExtEventClass extends EventClass
      implements ExtSubjObsGrp. MyEvent {
   String getItem(){ … }
}
```

# More General Inheritance

- Details of type rules unimportant -- type-safe.
  - Need "exact" type-groups when send "binary" messages
- Inheritance effective.
- What about subtyping and/or matching?
  - How do they change?

# Subtyping & Inheritance?

- Can have one class/interface extend another within same group:
  - interface MyExtSubject extends MySubject{ … }
- Can insist this be preserved in extensions of groups.
- Parametric polymorphism:

```
class Test<SOG extends SubjObsGrp>
{ public void doIt( @SOG.MySubject sub){
   }
   … SOG.MyExtSubject sub' …
}
```

# Inheritance vs. Subtype & ??

- Inheritance: Modify classes (simultaneously) to obtain modified or added behavior so can still interact safely according to their signatures.
- Subtype - masquerading (what contexts can it be used in) - *no matter how definition given*.
  - More local?
- Matching - capabilities (what messages can it accept).
- What else? Wild cards??

*Questions?*