# Integrating Nominal and Structural Subtyping

**Donna Malayeri**
Jonathan Aldrich

## Structural vs. nominal subtyping

**Nominal Subtyping**

- A type *T* is a subtype of *U* only if *T* has been *declared* as a subtype of *U*
- The norm in mainstream languages like Java

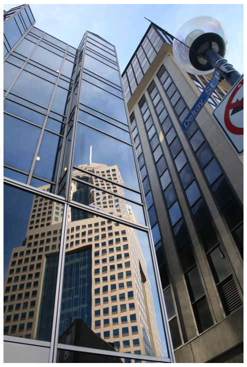
**Structural subtyping**

- a type *T* is a subtype of *U* if *T has at least U's methods and fields*—possibly more, possibly with more refined types
  - So, any class with an `iterator()` method would automatically be a subtype of `Iterable`

## Our language: Unity

- A type has:
  - a nominal component (a brand)
  - a structural component (its fields and methods)
- Subtyping takes both components into account
- Allows structural subtyping to co-exist with external dispatch
  - Combination is novel

## Why structural subtyping?



## A motivating example (Java)

```
interface Drawable {
   void draw();
   void setBounds(Rect bounds);
   void setAlpha(int alpha);
}
```

```
class Circle implements Drawable {
   void draw() { … }
   void setBounds(Rect r) { … }
   void setAlpha(int alpha) { … }
}
```

```
class Icon {
   void draw() { … }
   void setBounds(Rect r) { … }
}
```

```
void centerAndDraw(□ item)  {
    ... // compute rect
    item.setBounds(rect);
    item.draw();
}
```

## Our solution: Unity

```
type Drawable =
 Object (
   draw() : unit,
   setBounds(bounds:Rect) : unit,
   setAlpha(alpha:int): unit )
```

```
type Bitmap =
 Object (
   draw() : unit,
   setBounds(bounds:Rect) : unit)
```

```
brand Circle extends Object (
  method draw() : unit = …,
  method setBounds(r:Rect) = …,
  method setAlpha(alpha:int) = …
)
```

```
brand Icon extends Object (
  method draw() : unit = …,
  method setBounds(r:Rect) = …
)
```

- Structural subtyping: `Drawable ≤ Bitmap`
  `Circle ≤ Bitmap`
  `Circle ≤ Drawable`
  `Icon   ≤ Bitmap`

# Our solution: Unity

```
type Drawable =
 Object (
   draw() : unit,
   setBounds(bounds:Rect) : unit,
   setAlpha(alpha:int): unit )
```

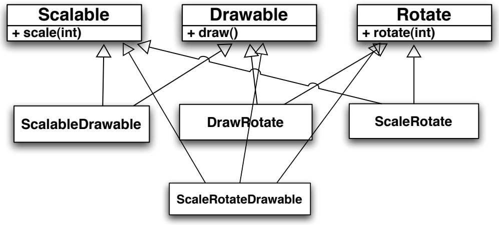
```
type Bitmap =
 Object (
   draw() : unit,
   setBounds(bounds:Rect) : unit)
```

```
brand Circle extends Object (
  method draw() : unit = …,
  method setBounds(r:Rect) = …,
  method setAlpha(alpha:int) = …
)
```

```
brand Icon extends Object (
  method draw() : unit = …,
  method setBounds(r:Rect) = …
)
```

```
method centerAndDraw(item : Bitmap) =
    ... // compute rect
    item.setBounds(rect);
    item.draw();
```

# Example 2: composing interfaces



```
class Glyph implements            {
  ...
}

void doSomething(ScaleRotate shape)
doSomething(new Glyph());
```

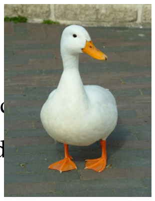
Method call fails!

# How to solve this problem?

- Problem: nominal subtyping doesn't compose
  - types `Scalable` and `Movable` do not compose to `ScalableMovable`
- But types DO compose in structural subtyping!
  - `{scale()}` and `{move()}` compose naturally to `{scale(), move()}`
- No need to manually define all combinations of types!

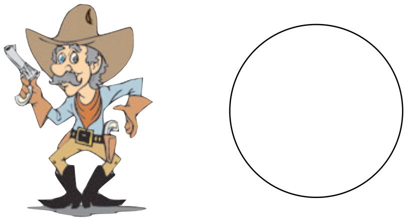
# Benefits of structural subtyping

- Flexible and compositional
- Allows unanticipated reuse
- No unnecessary proliferation of
- Useful for data persistence and d computing
- Examples: O'Caml objects, static "duck typing"

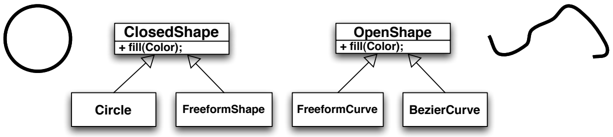


# Why nominal subtyping?

# Expressing intent



## Nominal subtyping benefits



- ClosedShape has the same interface as OpenShape, but we don't want them to be interc

```
void Image.mask(ClosedShape shape) {

myimage.mask(freeformCurve);  // type err
myimage.mask(circle);         // ok
```

## Additional benefits

Nominal Subtyping:

- Provides better error messages
- Facilitates natural and efficient external methods
  - More on this later
- Languages: Java, C#, C++, VB, Modula-3, etc.

## Solution: Unity

- Combines nominal and structural subtyping
- The *flexibility* and *composability* of structural subtyping
- Along with the *design intent* of nominal subtyping
- Types have ***both*** a nominal and structural component
- $A \le B$ iff
  $A \le_{\text{nominal}} B$ and $A \le_{\text{structural}} B$

## Example 3 in Unity

```
brand ClosedShape extends Object (...)

brand Circle extends ClosedShape (
   method fill() : unit = ... , ...)

brand FreeformCurve extends OpenShape (
   method fill() : unit = ... , ...)

brand Image extends Object (
   method mask(shape:ClosedShape) = ...
)


myimage.mask(freeformCurve);  // type error, FreeFormCurve ≰ ClosedShape
myimage.mask(circle);         // ok
```

## Example 3 in Unity

```
brand ClosedShape extends Object (...)

brand Circle extends ClosedShape (
   method fill() : unit = ... , ...)

brand FreeformCurve extends OpenShape (
   method fill() : unit = ... , ...)

brand Image extends Object (
   method mask(shape:ClosedShape(□)) = ...
)


myimage.mask(freeformCurve);  // type error, FreeFormCurve ≰ ClosedShape
myimage.mask(circle);         // ok
```

## Example 3 in Unity

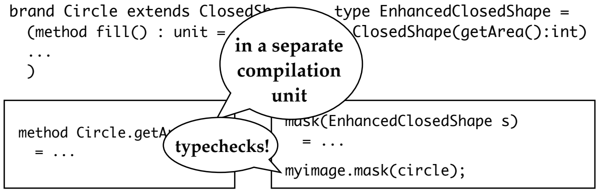
```
brand ClosedShape extends Object (...)

brand Circle extends ClosedShape (
   method fill() : unit = ... , ...)

brand FreeformCurve extends OpenShape (
   method fill() : unit = ... , ...)

brand Image extends Object (
   method mask(shape:ClosedShape(getArea():int)) = ...
)


myimage.mask(freeformCurve);  // type error, FreeFormCurve ≰ ClosedShape
myimage.mask(circle);         // type error, Circle lacks getArea() method
```

## Adding methods to implement an interface

```
brand Circle extends ClosedShape     type EnhancedClosedShape =
  (method fill() : unit = …            ClosedShape(getArea():int)
   ...
  )
```

- Want to add new method to `Circle` to make it implement `EnhancedClosedShape`
  - But, can't change `Circle` directly
- Solution: structural subtyping & external methods

## Structural subtyping + external methods



```
brand Circle extends ClosedSh           type EnhancedClosedShape =
  (method fill() : unit =                 ClosedShape(getArea():int)
  ...
  )
```

```
method Circle.getA                       mask(EnhancedClosedShape s)
  = ...                                    = ...

                                         myimage.mask(circle);
```

- External methods let you add methods to a brand, outside its definition
- Now `Circle` is structurally a subtype of `EnhancedClosedShape`

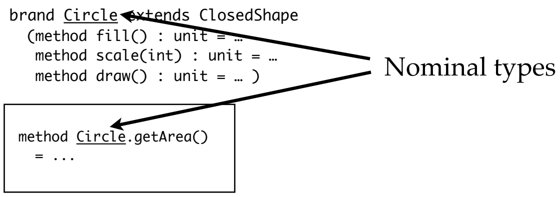
## External dispatch may cause ambiguity

- Non-example, structural dispatch:

```
type Foo = Object({foo:int})
type Bar = Object({bar:char})
method Foo.m() : unit = …
method Bar.m() : unit = …
```

- Inefficient: would have to check entire structure of type
- Ambiguous: what if m's receiver has type `{foo:int, bar:char}`?
- Because `{foo:int, bar:char} ≤ Foo`
  `{foo:int, bar:char} ≤ Bar`

## What are we dispatching on?



```
brand Circle extends ClosedShape
  (method fill() : unit = …
   method scale(int) : unit = …
   method draw() : unit = … )
```

```
method Circle.getArea()
  = ...
```

- Dispatch on *nominal* types (i.e. brands)
- Another reason to combine structural and nominal subtyping: external dispatch depends on nominal types!

## External methods in Unity

- Conceptually part of an existing brand/class
- Performs dispatch on objects of that brand's type
- Dispatch: method is selected based on the run-time type of the object
- Doesn't have to be in the same compilation unit as the brand

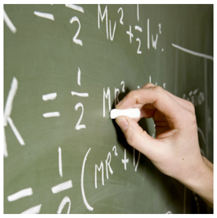
## External methods in Unity

- Conceptually part of an existing brand/class
- Performs dispatch on objects of that brand's type
- Dispatch: method is selected based on the run-time type of the object
- Doesn't have to be in the same compilation unit as the brand

# Unity benefits

- Makes it easier to maintain software, both in terms of ***interfaces*** and ***code***
- Structural subtyping eases the task of ***expressing an interface***
  - An interface is just a type and does not need to be declared in advance
- Nominal subtyping ***captures intent***
- External dispatch eases the task of ***conforming to an interface***

# Examples



# Eclipse JDT: example 1

- All of these classes have method `IBinding resolveBinding()`

  ```
  ImportDeclaration
  MemberRef
  MethodRef
  Name
  AnnotationTypeDeclaration
  AnonymousClassDeclaration
  EnumDeclaration
  Type
  ```
  ***… plus 8 more***
- But there's no `HasBinding` interface with a `resolveBinding()` method
- Structural subtyping would solve this problem—just declare the interface after-the-fact

# Eclipse JDT: example 2

- All of these classes have method `SimpleName getName()`

  ```
  AbstractTypeDeclaration
  AnnotationTypeMemberDeclaration
  EnumConstantDeclaration
  FieldAccess
  MemberRef
  MemberValuePair
  MethodDeclaration
  MethodInvocation
  ```
  ***… plus 8 more***
- But there's no `HasName` interface with a `getName()` method

# Displaying elements in a tree view: Java

```
class MyLabelProvider extends LabelProvider
{
  String getText(Object element) {
    String label;
    if (element instanceof AbstractTypeDeclaration)

   if (element instanceof AbstractTypeDeclaration)
          label = ((AbstractTypeDeclaration) element).
                      getName().toString();

    else if (element instanceof FieldAccess)
        label = ((FieldAccess) element).
                getName().toString();
    else if (element instanceof MemberRef)
        label = ((MemberRef) element).
                getName().toString();
    ...

    return label;
  }
}
```

# Displaying elements in a tree view: Unity

```
brand MyLabelProvider extends LabelProvider (
    method getText(element : Object(getName() : SimpleName)) : String =
       element.getName().toString()
}
```

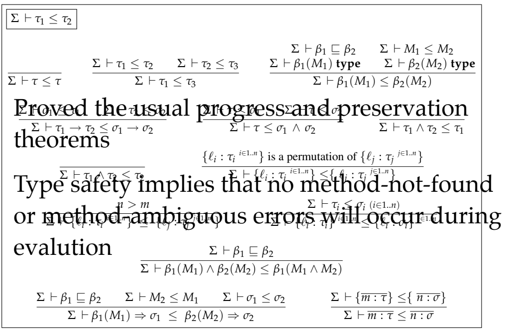
# Empirical evidence

- Empirical study of 15 Java applications showed that 12%-28% of methods share a name but not a common supertype
  - Range from 164 to 24,500 methods in application
  - Example: 5 iterator decorators in Apache Collections have methods `getIterator` and `setIterator`

# Summary of results

| | Total methods | %common methods |
|---|---|---|
| **Tomcat** | **14678** | **28.4%** |
| **Ant** | **9178** | **28.1%** |
| **JHotDraw** | **5149** | **23.2%** |
| **Smack** | **3921** | **22.5%** |
| **Struts** | **3783** | **20.4%** |
| **Apache Forrest** | **164** | **17.1%** |
| **Cayenne** | **9243** | **16.7%** |
| **Log4j** | **1950** | **16.0%** |
| **OpenFire** | **8135** | **16.0%** |
| **Apache Collections** | **3762** | **15.5%** |
| **Derby** | **24521** | **14.6%** |
| **Lucene** | **2472** | **13.4%** |
| **jEdit** | **5845** | **12.0%** |
| **Apache HttpClient** | **1818** | **11.9%** |
| **Areca** | **3565** | **11.9%** |

# Type soundness proof



$$\boxed{\Sigma \vdash \tau_1 \le \tau_2}$$

$$\frac{}{\Sigma \vdash \tau \le \tau} \qquad \frac{\Sigma \vdash \tau_1 \le \tau_2 \qquad \Sigma \vdash \tau_2 \le \tau_3}{\Sigma \vdash \tau_1 \le \tau_3} \qquad \frac{\Sigma \vdash \beta_1 \sqsubseteq \beta_2 \qquad \Sigma \vdash M_1 \le M_2 \qquad \Sigma \vdash \beta_1(M_1)\ \textbf{type} \qquad \Sigma \vdash \beta_2(M_2)\ \textbf{type}}{\Sigma \vdash \beta_1(M_1) \le \beta_2(M_2)}$$

$$\frac{\Sigma \vdash \sigma_1 \le \tau_1 \qquad \Sigma \vdash \tau_2 \le \sigma_2}{\Sigma \vdash \tau_1 \to \tau_2 \le \sigma_1 \to \sigma_2} \qquad \frac{\Sigma \vdash \tau \le \sigma_1 \qquad \Sigma \vdash \tau \le \sigma_2}{\Sigma \vdash \tau \le \sigma_1 \wedge \sigma_2} \qquad \frac{}{\Sigma \vdash \tau_1 \wedge \tau_2 \le \tau_1}$$

$$\frac{}{\Sigma \vdash \tau_1 \wedge \tau_2 \le \tau_2} \qquad \frac{\{\ell_i : \tau_i{}^{i\in 1..n}\} \text{ is a permutation of } \{\ell_j : \tau_j{}^{j\in 1..n}\}}{\Sigma \vdash \{\ell_i : \tau_i{}^{i\in 1..n}\} \le \{\ell_j : \tau_j{}^{j\in 1..n}\}}$$

$$\frac{n > m}{\Sigma \vdash \{\ell_i : \tau_i{}^{i\in 1..n}\} \le \{\ell_j : \tau_j{}^{j\in 1..m}\}} \qquad \frac{\Sigma \vdash \tau_i \le \sigma_i \ (i\in 1..n)}{\Sigma \vdash \{\ell_i : \tau_i{}^{i\in 1..n}\} \le \{\ell_i : \sigma_i{}^{i\in 1..n}\}}$$

$$\frac{\Sigma \vdash \beta_1 \sqsubseteq \beta_2}{\Sigma \vdash \beta_1(M_1) \wedge \beta_2(M_2) \le \beta_1(M_1 \wedge M_2)}$$

$$\frac{\Sigma \vdash \beta_1 \sqsubseteq \beta_2 \qquad \Sigma \vdash M_2 \le M_1 \qquad \Sigma \vdash \sigma_1 \le \sigma_2}{\Sigma \vdash \beta_1(M_1) \Rightarrow \sigma_1 \ \le \ \beta_2(M_2) \Rightarrow \sigma_2} \qquad \frac{\Sigma \vdash \{\overline{m : \tau}\} \le \{\ \overline{n : \sigma}\}}{\Sigma \vdash \overline{m : \tau} \le \overline{n : \sigma}}$$

- Proved the usual progress and preservation theorems
- Type safety implies that no method-not-found or method-ambiguous errors will occur during evalution

# Selected Related Work

- Similar approaches after our initial proposal:
  - Scala [Odersky '07], Whiteoak [Gil and Maman '08] *not formalized*
- External methods: MultiJava [Clifton et al '00]
- Only structural *typing*, not subtyping: Modula-3

# Summary

- Unity combines structural and nominal subtyping
- Allows structural subtyping to co-exist with external dispatch
  - Each adds flexibility to the language
  - Combination is novel
- Evidence that existing programs could benefit