Review: Subtyping rules

In the type system:
- Subtyping relation: Write \( t_1 <: t_2 \) for \( t_1 \) is a subtype of \( t_2 \)
- One new typing rule that uses subtyping:
  - If \( \alpha \) has type \( t_3 \) and \( t_1 <: t_2 \), then \( \alpha \) (also) has type \( t_2 \)

Define \( t_1 <: t_2 \):
- Principle of substitutability: if \( t_1 <: t_2 \), then any value of type \( t_1 \) must be usable in any way a \( t_2 \) is.
- Transitivity: if \( t_1 <: t_2 \) and \( t_2 <: t_3 \), then \( t_1 <: t_3 \)
- Reflexivity: Every type is a subtype of itself, i.e., \( t <: t \)

Review: Subtyping rules

- **Width subtyping:**
  - If \( t_1 \) has more fields than \( t_2 \), but otherwise agrees on field types, then \( t_1 <: t_2 \)
    - A record subtype can have more fields than its supertype
- **Depth subtyping:**
  - If \( t_a <: t_b \), and \( f \) is immutable, then
    \[
    \{f_1:t_1, \ldots, f:t_a, \ldots, f_n:t_n\} <: \{f_1:t_1, \ldots, f:t_b, \ldots, f_n:t_n\}
    \]
    - Mutable fields cannot have their type change by subtyping
  - Invariant subtyping: neither contra- nor covariant
- **Function subtyping:**
  - If \( t_3 <: t_1 \) and \( t_2 <: t_4 \), then \( t_1 -> t_2 <: t_3 -> t_4 \)
    - Contravariant in argument(s) and covariant in results

Now...

Use what we learned about subtyping for records and functions to understand subtyping for class-based OOP
- e.g., Java and C#

One big change:
- Class names are also types
- Subclasses are also subtypes

Objects encoded as records

- Objects: essentially records where some fields are...
  - The “normal” fields we think of
  - But others are the methods: functions that have access to \( \texttt{self} \)
  
  ```scala
  val pt = (x = 1.0, y = 2.0, 
  distToOrigin = fn self => Math.sqrt(self.x*self.x + self.y*self.y))
  val d = pt.distToOrigin(pt)
  ```
  - Methods are usually immutable fields, but don’t have to be (e.g., Ruby)
  - But that encoding requires callers to always pass in the right receiver...
Objects encoded as records

- MF convention: compiler always inserts receiver as implicit argument

- That’s what OOP languages actually do
- So could design a type system using types very much like record types...

```
val pt = { x = 1.0, y = 2.0, distToOrigin = fn () => Math.sqrt(self.x*self.x + self.y*self.y) }
val d = pt.distToOrigin()
```

Subtyping for objects in MF

- A subtype of an object type can have extra fields [width subtyping]
- Mutable fields of an object type must have same type (invariant) in any subtypes [depth subtyping on mutable fields]
- A subtype of an object type can have extra methods [width subtyping]
- Method fields are immutable, and a method of a subtype can have covariant return type and contravariant argument types [depth subtyping on immutable fields, and function subtyping]

```
type point = { x:real, y:real, move_right: real->point }
val pt:point = { x = 1.0, y = 2.0, distToOrigin = fn () => Math.sqrt(self.x*self.x + self.y*self.y) };
val d = pt.distToOrigin()
```

Subtyping for objects in Java

- In Java (and C#), types and classes are conflated:
  - Declaring class also declares type with same name
  - class C {...} creates a type C
  - Declaring a subclass also declares a subtype
    - C extends D causes C <: D
    - C implements D causes C <: D
  - So we don’t have to write down the “complicated” MF record types

```
class Pt {
  double x, y;
  double distance(Pt s) { ... } // Pt shift(double dx, double dy) { ... }
  ... // Function
}
```

Actual Java

```
class Pt {
  double x, y;
  double distance(Pt s) { ... } // Pt shift(double dx, double dy) { ... }
  ... // Function
}
```

```
interface Colorable {
  Color getColor();
  void setColor(Color c);
}
```

```
class ColorPt extends Pt implements Colorable {
  Color color;
  Color getColor () { return this.color; }
  void setColor(Color c) { this.color = c; }
  ColorPt shift(double dx, double dy) {
    Pt p = super.shift();
    return new ColorPt(p.x,p.y,this.color);
  }
}
```

Example (constructors and public omitted)

```
class Pt {
  double x, y;
  double distance(Pt s) { ... } // Pt shift(double dx, double dy) {
  ... // Function
}
```

```
interface Colorable {
  Color getColor();
  void setColor(Color c);
}
```

```
class ColorPt extends Pt implements Colorable {
  Color color;
  Color getColor () { return this.color; }
  void setColor(Color c) { this.color = c; }
  ColorPt shift(double dx, double dy) {
    Pt p = super.shift();
    return new ColorPt(p.x,p.y,this.color);
  }
}```
Example so far

- An instance of ColorPt is substitutable for any value of type Pt or type Colorable
  - Adds field color
  - Gives shift a more specific return type
  - Adds methods w.r.t. ColorPt and w.r.t. Colorable

- What about changing the types of fields or method arguments…?

Field shadowing

- Mutable fields must have the same type in subclass and superclass, so no "overriding" possible
  - Allowing Rhino to sit on Zebra would be unsound
- Java instead says: a field declared in the subclass can have the same name as an inherited field, but it is a new, different field
  - Field in subclass shadows
  - Can access other field with super.f
  - No dynamic dispatch: inherited methods use old field

Static overloading

- A Java class can have multiple methods with the same name
- So must again revisit the key question in OOP: What does e0.m(e1,...,en) mean?
- As before:
  - Evaluate e0,...,en to o0,...,on
  - Look up class of o0 (dynamic dispatch)
- But now the class may have more than one m
  - Java: Pick the "best" one using the static types of e1,...,en
  - The (run-time) class of o1,...,on is irrelevant
  - "Best" is complicated, roughly "least amount of subtyping"

Change type of field?

```java
class Animal
class Zebra extends Animal {
    void showStripes () {...} }
class Rhino extends Animal { }
class Zoo { Animal a; }
class ZebraZoo extends Zoo { Zebra a; }
class RhinoZoo extends Zoo { Rhino a; }
void installRhino(Zoo z) {
    z.a = new Rhino();
}
z = new ZebraZoo();
z.a = new Zebra();
installRhino(z); // A Rhino just sat on a Zebra!
// So is Java unsound?
```

Change type of method?

- Immutable methods could have covariant return types and contravariant argument types
  - Java allows the covariant returns
  - But not the contravariant arguments
- Java instead says: a method declared with different argument types is a different method with the same name
  - Simply no syntax for overriding with contravariant args
  - Result is static overloading...

Static overloading examples

```java
class Color extends Object { String s; }
class FancyColor extends Color { double shade; }
class MyClass {
    void m(Object x) { ... } // A
    void m(Color x) { ... } // B
    void m(FancyColor x) { ... } // C
    void m(Color x, FancyColor y) { ... } // D
    void m(FancyColor x, Color y) { ... } // E
}
MyClass obj = new MyClass();
Color cl = new Color(...);
FancyColor cf = new FancyColor(...);
Color cl = new FancyColor(...); // subtyping!
obj.m(cl), // B
obj.m(cl); // C
obj.m(cl), // B static overloading!
obj.m(cl, cl); // D
obj.m(cl, cl); // type error: no method matches
obj.m(cl, cl); // type error: no best match (tie)
```
Static overloading

- Static overloading saves you the trouble of making up different method names
  - Rational class could have add methods for
    - Rational
    - int
    - double
    - etc.
  - Often convenient, but the exact rules are complicated
- Static overloading is not multimethods
  - Multimethods look up method using run-time class of all args
  - So still have to code up double dispatch manually

self/this is special

- Our MF encoding of objects made self/this an argument to methods
- So if self/this is a function argument, is it contravariant?
  - No, it is covariant: a method in a subclass can use fields and methods only available in the subclass: essential for OOP

Mini-review

- Java’s rules for subtyping are sound
  - based on sound record and function subtyping rules
  - but allow even less subtyping than those rules would
- Biggest unnecessary restriction in Java is having subtyping only via subclasses and interfaces...

Names vs. structure

- From a "method not understood" perspective, no reason we couldn’t make

  ```java
  class ThreeActPlay <: StringPair
  {
  String first;
  String second;
  String third;
  void setFirst(String x){ ... }
  }
  ```

  ```java
  class ThreeActPlay
  {
  String first;
  String second;
  String third;
  void setFirst(String x){ ... }
  }
  ```

  - Silly example, but key idea behind duck-typing: Is the type of an object "what it can do" or "its place in the class hierarchy"
    - Interfaces the former, but require explicit implements clause

Classes vs. types

- A class defines an object’s behavior
- Subclassing modifies run-time behavior by extension and overriding
- A type defines what fields an object has and what messages it can respond to
- Subtyping determines compile-time behavior by defining when one value is soundly substitutable for another

- These are separate concepts: try to use the terms correctly
  - Java/C# confuse them by conflating classes and types
  - This confusion is convenient in practice