Static typing in object-oriented languages

Keunwoo Lee
CSE 341 -- Programming Languages
University of Washington
Dept. of Computer Science and Engineering

Static types: review

- Need to statically eliminate "unsafe" operations
  - (undecidable in general case; use conservative approximation)
- **"Unsafe"**: relative to definition of language

- In OO languages: generally "unsafe" = sending message to object that has no method for it
  - **"message not understood"** exception
  - static type system guarantees no "message not understood" exceptions

Typing OO programs

- Assign type to every expression

1 For every message send: make sure type of receiver contains method for message send (name and argument types)

2 For every method body, ensure it returns correct type (assuming types of args & receiver)

3 Every class must implement types it declares

4 Every class must be compatible extension of its superclass

Terminology

- **class:** unit of **implementation**
  - instructs **compiler** how to generate code
  - mostly concerns **dynamic** semantics

- **type:** unit of **interface**
  - instructs **type checker and programmer** how an expression may be used
  - mostly concerns **static** semantics

Object type syntax

- **object types are like record types: a map from names to types**
- **Could use ML type syntax:**

```
{ fieldName1:type1,
  ...,
  fieldNameN:typeN,
  methodName1:argType1 -> returnType1,
  ...
  methodNameM:argTypeM -> returnTypeM }
```

Object type syntax (2)

- **Instead, we'll use more familiar Java-like syntax:**

```
signature S {
  type1 fieldName1;
  ...
  typeN fieldNameN;
  returnType1 methodName1(argType, ..., argType);
  ...
  returnTypeM methodNameM(argType, ..., argType);
}
```

Object type example

```
signature Point {
    Integer x;
    Integer y;
    Point move(Integer dx, Integer dy);
}
```

- Ignore access protection for now --- all public
- Recall types describe only interface --- no bodies
- Will sometimes omit signature name (Point)
- Can permute members at will (order does not matter)

Fields = methods

- **Read-only field is equivalent to method:**

  signature { Foo x; }

  is equivalent to

  signature { Foo x(); }

- **Read-write field is equivalent to two methods:**

  signature { mutable Foo x; }

  is equivalent to

  signature { Foo x(); void setFoo(Foo x); }

- **Will mostly ignore fields in discussion that follows**
- **Rules for fields can be derived straightforwardly from rules for methods.**

Subtyping

- Subtyping is essence of OO types
- T1 subtypes T2 if instances of T1 can be substituted for instances of T2
  - i.e., T1 understands all messages of T2, and always returns type-compatible results
  - "Substitutability principle"
- Notation: "T1 subtypes T2" written T1 <: T2

Reflexive, transitive

- All types subtype themselves:

  T <: T   (reflexivity)

- Subtyping is transitive:

  T1 <: T3   and   T3 <: T2

  implies

  T1 <: T2

Width subtyping

- If T1 has exactly the same members as T2, plus some extra ones, then T1 <: T2

```
signature Point {
  Integer x();
  Integer y();
  Point move(Integer dx, Integer dy);
}
signature ColoredPoint {
  Integer x();
  Integer y();
  Color color();
  Point move(Integer dx, Integer dy);
}
```

  - Can derive **ColoredPoint <: Point**

Depth subtyping

- **If T1 is exactly like T2, except that one of T1's methods subtypes one of T2's methods, then T1 <: T2.**

```
signature Rectangle {
  Point topLeft();
  Point bottomRight();
}
signature ColoredRectangle {
  ColoredPoint topLeft();
  ColoredPoint bottomRight();
}
```

  - ColoredRectangle substitutable for Rectangle --- result of topLeft() always substitutable

Method subtyping

- But hold on --- depth subtyping asks whether methods subtype each other
- Must define *method* subtyping relation...
- (trickier than it seems)

Fruits, plants, flies

```
signature Fruit  { String name(); }
signature Apple  { String name(); Stem stem(); }
signature Banana {
    String name(); void slipOnPeel(); }

signature FruitPlant { Fruit produce(); }
signature ApplePlant { Apple produce(); }

signature FruitFly { void eat(Fruit f); }
signature AppleFly { void eat(Apple a); }
```

Fruit subtyping

```
signature Fruit  { String name(); }
signature Apple  { String name(); Stem stem(); }
signature Banana {
    String name(); void slipOnPeel(); }
```

- **Seems clear that**

  Apple <: Fruit

  Banana <: Fruit

- Indeed, width subtyping gives us this result

Return subtyping

signature FruitPlant { Fruit produce(); }

signature ApplePlant { Apple produce(); }

Seems OK to conclude that

ApplePlant <: FruitPlant

Result of produce() always substitutable:

```
ApplePlant ap = ...;
FruitPlant fp = ap;
Fruit f = fp.produce();
String s = f.name();
```

- **Return types are *covariant* (go *with* subtyping relationship of method as a whole)**

Argument subtyping

signature FruitFly { void eat(Fruit f); }

signature AppleFly { void eat(Apple a); }

Can we conclude that

AppleFly <: FruitFly ?

Consider following code:

```
AppleFly af = ...;        // 1
FruitFly ff = af;         // 2
Fruit aFruit = ...;       // 3
ff.eat(aFruit);           // 4
```

What if the AppleFly implementor calls stem() on its argument?

"Natural" subtyping

- **Covariant *argument* subtyping is broken!**
- **Must use opposite rule --- called *contravariant* rule --- for arguments.**

- Summary:
  - **For M1 to subtype M2, M1 must *return* a type *at least as specific as* M2.**
  - **For M1 to subtype M2, M1 must *accept argument types* that are *at least as general as* M2's.**

Other rules...

- **Java uses *invariant* argument and return:**
  - M1 subtypes M2 only if M1 and M2 have *same* argument and return types.

- **C++ uses *invariant* argument and *covariant* return:**
  - M1 subtypes M2 only if M1 and M2 have *same* argument types, and M1's return type is *at least as specific* as M2's

- **Eiffel uses *covariant* argument and return types**
  - M1 subtypes M2 only if M1's argument and return types are *at least as specific* as M2's.
  - Broken! (Fix using dynamic checks: raise runtime error)

Implementations

```
class C1
  subclasses C2
  implements S1, S2, ... SN
{
  returnType1 methodName1(argType, ... argType)
     { body1 }
  ...
  returnTypeN methodNameM(argType, ... argType)
     { bodyM }
}
```

Completeness

Completeness of implementation rule:

- A class C must have a method --- either defined in C, or inherited from C's superclass(es) --- to handle every message in its types.

```
class MauvaisePomme
    subclasses Object
    implements Apple {
    String name() { return "BadApple"; }
}

MauvaisePomme mp = ...;   // 1
Apple a = mp;             // 2
Stem s = a.stem();        // 3
```

Abstract classes

- Most languages allow **abstract methods**
- Classes that do not implement all methods in their types, or that do not override abstract methods with non-abstract ones, are **abstract classes**

- **Concrete instantiation restriction:**
  - Only non-abstract classes can be instantiated.

- Note this relaxes completeness of implementation rule --- incomplete classes exist, but may not be instantiated

Compatible extension

```
class BonFruit subclasses Object implements Fruit {
   String name() { return "some kind of fruit"; } }

signature Bogus { Integer name(); }
class Papaya subclasses BonFruit implements Bogus {
   Integer name() { return 456; } }
```

- Problem: most languages require that subclasses also be supertypes
- In such languages, methods must override only with a method that subtypes overridden method

Miscellaneous issues

- Access protection
- Structural vs. nominal subtyping
- Principal typing of classes
- Overloading vs. overriding
- Subtyping of mutable objects

Access protection

- **To add access protection (public, private, protected):**
  - Add visibility modifiers to fields and methods
  - Change typechecking of sends, classes, inheritance

- Won't discuss details in this class
- Recall that in ML we use module system to accomplish much the same thing --- arguably a more orthogonal design (does not conflate data type with module)

By-name subtyping

- **Our presentation has used *structural* subtyping**
- **Most real-world languages use *by-name* (nominal) subtyping:**
  - T1 subtypes T2 if T1's structure subtypes T2,
    and
    T1 ***declares*** that it subtypes T2

  - e.g., following do not have subtype relation in Java:
    ```
    interface I1 { void foo(); }
    interface I2 { void foo(); void bar(); }
    ```

  - Must add:
    interface I2 **extends I1** { void foo(); void bar(); }

Principal class types

- **In Java, type checker implicitly declares a type for every class:**

```
class Point {
    Integer x() { ... }
    Integer y() { ... }
}
Point p = new Point( ... );
```

- Each class has **principal type**
  ("best type for that class")

Overloading

```
class Point extends Object {
    Integer x() { ... }
    Integer y() { ... }
    Point move(Integer dx, Integer dy) { ... }
    Point move(Float dx, Float dy) { ... }
}
```

Point move(Integer, Integer) and
Point move(Float, Float)
do not *not* have an overriding relationship --- they are different functions with the same name

Overloading ct'd

- Overloading resolves **statically**, based on **static type of arguments**, with surprising results:

```
class Shape extends Object {
  boolean overlaps(Shape other) { ... }
}
class Rectangle extends Shape {
  boolean overlaps(Shape other)     { ... }
  boolean overlaps(Rectangle other) { ... }
}
Rectangle r = new Rectangle(...);
Shape s = new Rectangle(...);
boolean b = r.overlaps(s);
```

Subtyping and mutation

```
signature FruitRef {
   Fruit fruit();
   void setFruit(Fruit f);
}

signature AppleRef {
   Apple fruit();
   void setFruit(Apple a);
}
```

Any subtype relation?

Subtyping & mutation (2)

Same with mutable fields...

```
signature FruitRef {
   mutable Fruit fruit;
}

signature AppleRef {
   mutable Apple fruit;
}
```

Subtyping & mutation (3)

```
class BananaImplementor
   extends Object
   implements Banana {
   String name() { ... }
   void slipOnPeel() { ... }
}

AppleRef ar = new AppleRefImplementor();        // 1
FruitRef fr = ar;                               // 2
fr.fruit = BananaImplementor();                 // 3
Apple anApple = ar.fruit;                       // 4
Stem s = anApple.stem();                        // 5
```