USER-ORIENTED LANGUAGE DESIGN

CS 2110 – Spring 2016

Decidability

Wildcards

```
public Variable {
    boolean value;
    /** Add this to the list corresponding to value */
    public void addTo(
            List<? super Variable> trues,
            List<? super Variable> falses) {
        (value ? trues : falses).add(this);
    }
}
```

Subtyping

Contrived

```
class C⟨P⟩ extends D⟨D⟨? super C⟨L⟨P⟩⟩⟩⟩ {}
```

C⟨X⟩ <: D⟨? super C⟨X⟩⟩

Inheritance

D⟨D⟨? super C⟨L⟨X⟩⟩⟩⟩ <: D⟨? super C⟨X⟩⟩

Instantiation

Infinite Proof of Subtyping

Inheritance

D⟨D⟨? super C⟨L⟨X⟩⟩⟩⟩ <: D⟨? super C⟨L⟨X⟩⟩⟩

Instantiation

C⟨L⟨X⟩⟩ <: D⟨? super C⟨L⟨X⟩⟩⟩

Efficiency

```
{
    interface Eq<in T>  {
        bool equalTo(T other); }

    interface List<out T> :
        Eq<List<Eq<T>>> { }

    interface Tree :
        List<Tree> { }
```

Restrictions

Termination Guaranteed

```
class C⟨P⟩ extends D⟨D⟨? super C⟨L⟨P⟩⟩⟩⟩ {}
```

Contrived

Inheritance Restriction

No use of ? super in the inheritance hierarchy

```
⟨P extends List⟨List⟨? super C⟨L⟨P⟩⟩⟩⟩⟩
```

Contrived

Parameter Restriction

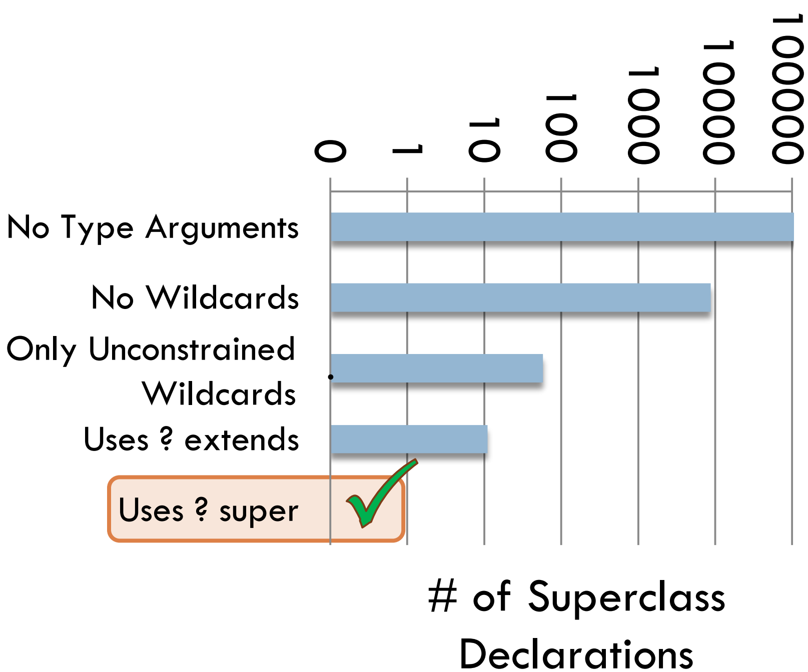
When constraining type parameters,
? super may only be used at covariant locations

Survey

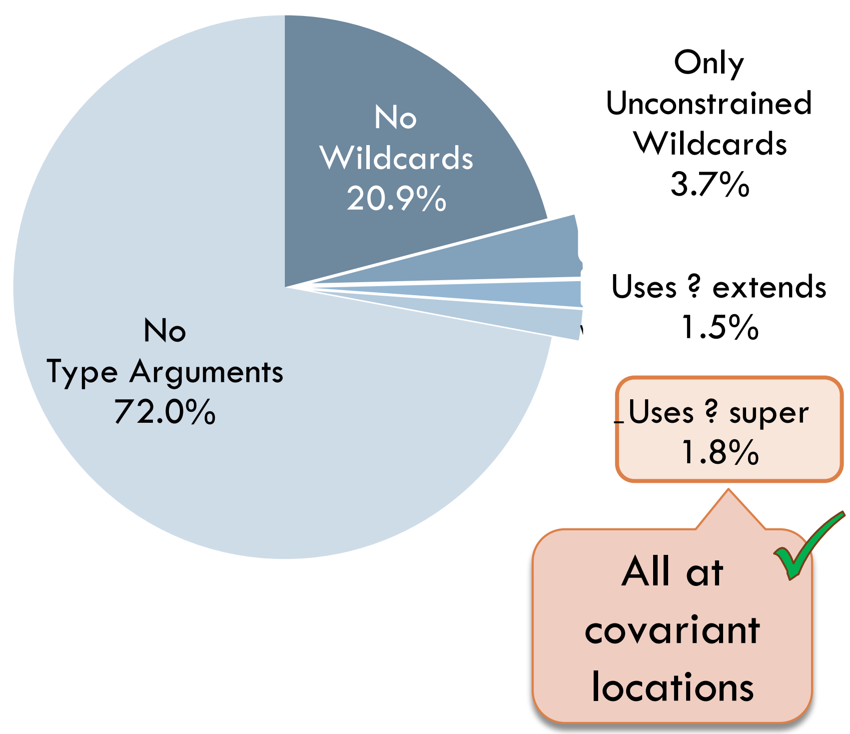
No Violations of Our Restrictions

9.2 Million Lines of Code Analyzed

Wildcards in Inheritance



Wildcards in Constraints



Industry Collaborations

Gavin King

at redhat on ceylon

Andrey Breslav

at JetBRAINS on Kotlin by JetBrains

Materials and Shapes

- Material
  - List, Integer, Property, Comparator
- Shape
  - Comparable, Summable, Cloneable

No class/interface is both a material and a shape

13.5 Million Lines of Code Analyzed

Programmers are Humans

Library Designer

- Want to provide a “separate” function
  - Inputs: middle, elems, smaller, bigger
  - Requirements:
    - Place all values in elems less than middle into smaller
    - Place all other values in elems into bigger
- Goals
  - Implement separate
  - Provide maximally flexible type signature

Library User

- Goal
  - Place nonnegative values in "ints" into "positives"
- Context
  - "ignore" throws away all elements added to it
- Implementation
  - separate(0, ints, ignore, positives);

User Types

- ints : Iterable<Integer>
  - You can get things from iterables
- positives : Collection<Integer>
  - You can add things to collections
- ignore : Collection<Object>
  - You can add anything to it
- Integer implements Comparable<Number>
  - integers can be compared with any number

Library Implementation

- void separate(middle,

```
void separate(middle,
              elems,
              smaller,
              bigger) {
    foreach (elem in elems)
              (elem < middle ? smaller : bigger)
                                  .add(elem);
}
```

Library Type

- <T extends Co

  void separ … able,

  T> elems,

  ec … > smalle

  lectio … igger

  forea … lem in e

  < middle … : bigger)

  ;

  }

Insufficient Flexibility

- **Formals**
- <T extends Comparable<T>>
  void separate(T middle,
  Iterable<T> elems,
  Collection<T> smaller,
  Collection<T> bigger)

Actuals

Integer	🚫
0	✓
ints	✓
ignore	🚫
positives	✓

Wildcards

- ```
  <T extends Comparable<? super T>>
  void separate(T middle,
                Iterable<? extends T> elems,
                Collection<? super T> smaller,
                Collection<? super T> bigger) {
      foreach (elem in elems)
                (elem < middle ? smaller : bigger)
                              .add(elem);
  }
  ```
  ```

Excessive Annotations

- <T>
  void flatten(
  Iterable~~<? extends~~ Iterable<~~? extends T~~>>
  colls,
  Collection<? super T> into) {
  for (Iterable<~~? extends T~~> coll : colls)
  for (T elem : coll)
  into.add(elem);
  }

Declaration-Site Variance

- <T extends Co
  void separa ... dle,
  T> elems,
  ec ... > smalle
  lectio ... igger
  forea ... lem in e
  < middle ... : bigger)
  ;
  }

Insufficient Flexibility

- **Formals**
- <T extends Comparable<T>>
  void separate(T middle,
  Iterable<T> elems,
  Collection<T> smaller,
  Collection<T> bigger)

Actuals

Integer	✓
0	✓
ints	✓
ignore	🚫
positives	✓

Declaration-Site Variance Retry

```
<T extends Comparable<T>,
            U super T, V super T>
void separate(T middle,
            Iterable<T> elems,
            Collection<U> smaller,
            Collection<V> bigger) {
    foreach (elem in elems)
            (elem < middle ? smaller : bigger)
                        .add(elem);
}
```

Mixed-Site Variance

- ```
  <T extends Comparable<T>>
  void separate(T middle,
                Iterable<T> elems,
                Collection<in T> smaller,
                Collection<in T> bigger) {
      foreach (elem in elems)
                (elem < middle ? smaller : bigger)
                              .add(elem);
  }
  ```
  ```

Use+Declaration-Site

- The two address orthogonal roles
  - declaration-site for class/interface designers
  - use-site for class/interface users
- How do the two interact?
  - given interface Iterator<out T> {…}
  - what does Iterator<in Number> mean?

Expectations vs. Designs

	Java	Scala	Default	Layer	Join	Mixed
Has declaration-site variance						
C<**τ**> <: C<out **τ**>						
C<out **τ**> <: C<out **τ'**> implies **τ** <: **τ'**						
instance of C<out τ> is instance of C<τ'> for some τ' <: τ						
instance of C<out τ> allocated as C<τ'> for some τ' <: τ						
C<in τ out τ'> is expressible						
C<in τ out τ> is valid implies C<τ> is valid						
Out<in τ> is valid						
implicit constraints used in subtyping						

Principal Types

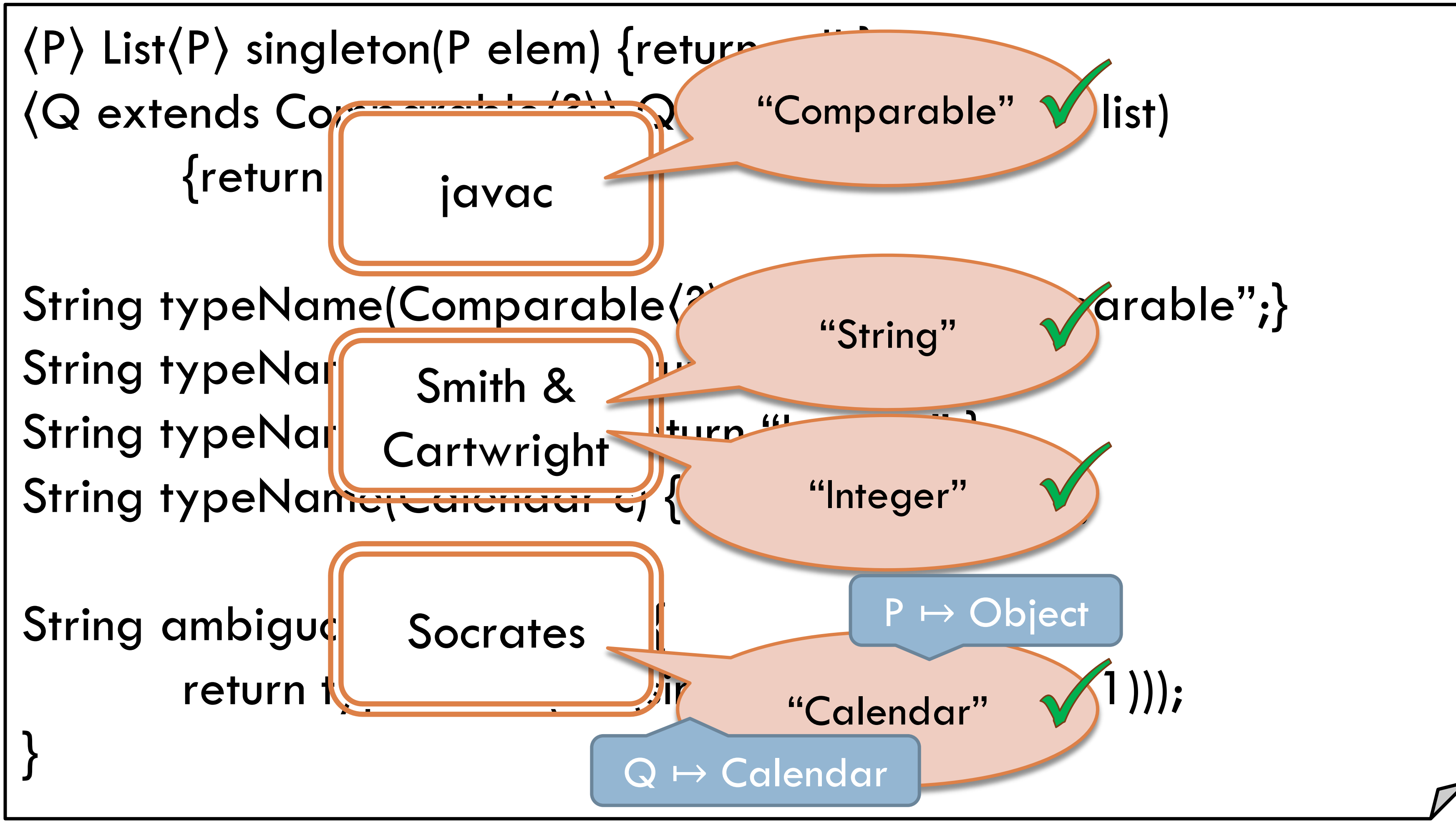
Principal Type

- The principal type of an expression
  - a type for that expression that is better than all other types for that expression
- "Hello" has principal type String
  - "Hello" also has type Object, CharSequence, …
  - String is a subtype of all those types
- A language has principal types
  - if every possible expression has a principal type

Java

- assertEquals(5, Integer.valueOf(5))
  - ambiguous!
  - Is it two ints or two Integers?
- But the expression 5 is also an Integer
- And Integer.valueOf(5) is also an int
- Neither expression has a principal type

Ambiguous Semantics



Use-Site Inferability Check

- ⟨T⟩ List⟨T⟩ singletonList(T) {…}
- var objs = singletonList(“Hello”);
- objs.add(5);
  - fails to type check
  - objs is inferred to be an List⟨String⟩
  - needs to be an List⟨Object⟩

Declaration-Site Inferability

- ⟨T⟩ List⟨T⟩ singletonList(T)
  - T is not inferable because Array is invariant
  - singletonList("Hello")
    - could have type List⟨String⟩ *or* List⟨Object⟩
    - no principal type
- ⟨T⟩ Iterable⟨? extends T⟩ singletonIterable(T)
  - T is inferable because ? extends is covariant
  - singletonIterable("Hello")
    - has type Iterable⟨? extends String⟩
    - which is subtype of Iterable⟨? extends Object⟩

Gradual Types

Goal

- Mix static and dynamic type systems
  - e.g. Java with JavaScript
- Requirements
  - no implicit insertions of wrappers
  - dynamic code is just static code minus types
  - stripping types preserves or improves semantics
  - static code can assume type annotations are true

C#'s `dynamic` Type

- ```
  bool Equal(object left, object right) {
      return left == right;
  }
  ```

- Equal(0, 0) returns false
  ```

C#'s `dynamic` Type

- interface Getter⟨T⟩ { T get(); }
- class Five : Getter⟨int⟩, Getter⟨string⟩ {
    int Getter⟨int⟩.get()
        { return 5; }
    double Getter⟨string⟩.get()
        { return 5.0; }
  }
- void Print(Getter⟨int⟩ getter) {
    Console.WriteLine(getter.get());
  }

Crashes if changed to dynamic!

C#'s `dynamic` Type

- ```
  List⟨T⟩ Snoc⟨T⟩(IEnumerable⟨T⟩ start,
                          T end) {
      var elems = ToList(start);
      elems.add(end);
      return elems;
  }
  ```
- Snoc(Singleton("Hello"), 5) works

Crashes if made dynamic!
  ```

Prerequisite Language Properties

- Static Behavioral Subtyping
  - Using a more precise type for a subexpression improves the typability of the whole expression
- Decidability
  - Typing must be reliably doable at run time
- Principality
  - Every execution has a most precise typing